Programmation Java

Notes de cours

Stéphane Airiau

L3 Informatique – MIDO
Année 2018–2019

DAUPHINE UNIVERSITÉ PARIS

PSL RESEARCH UNIVERSITY PARIS

Introduction

JAVA est un langage informatique : c'est un langage pour parler à une machine. Pour pouvoir écrire dans ce langage, il faut donc apprendre son vocabulaire et sa grammaire afin de pouvoir communiquer avec la machine.

Ce cours de mise à niveau ne suppose aucune connaissance particulière de JAVA . L'objectif du cours est de voir une grande partie des fonctionalités de ce langage. Le cours se déroule sur 15h avec une partie de cours et une partie de travaux dirigés. Ce temps n'est pas suffisant pour maîtriser toutes les subtilités du langage, mais il est suffisant pour pouvoir réaliser des projets qui peuvent déjà se montrer relativement complexes.

Pour ce cours, on fera l'hypothèse que des notions algorithmiques de base sont acquises. Ainsi, on ne va pas s'intéresser à l'implémentation d'algorithmes pour résoudre des problèmes particuliers et évaluer leurs performances d'un point de vue théorique (déterminer le nombre d'opérations pour mener à bien un calcul, ou bien évaluer les demandes en mémoire, etc). Ainsi, on ne s'intéressera pas à des exemples classiques comme le problème de tri d'un ensemble fini ou le calcul du chemin minimum entre deux noeuds d'un graphe. Le but du cours est de connaître les *fonctionalités* du langage. Par exemple, si on possède une implémentation d'un algorithme qui tri des nombres entiers, comment est-il possible de modifier ce code (et a-t-on besoin de modifier ce code) pour trier des étudiants selon leur moyenne ou selon leur pointure de chaussures.

Lorsqu'on implémente un projet dans un langage, il faut non seulement continuer à avoir ces exigences algorithmiques, mais il faut aussi penser à d'autres aspects, par exemple

— lisibilité : il faut toujours avoir en tête que chaque ligne de code écrite sera lu par quelqu'un d'autre : par un autre programmeur qui travaille sur le même projet, par quelqu'un qui va améliorer le projet, ou bien par vous-même quelques semaines après avoir écrit le code. Il faut donc écrire du code qui soit compréhensible par un lecteur : il faut penser à commenter, c'est à dire laisser un texte qui donne une intuition, une justification, des explications sur les lignes de codes qui suivent. Le choix du nom des variables participe également à la lisibilité du code.

— ré-utilisation/abstraction : même si chaque projet à un but précis, des parties du code, ou le projet lui-même peuvent être utilisés à d'autres fins. On peut donc ré-utiliser des parties de code et ne pas ré-inventer la roue à chaque projet. Par exemple, on implémente un code pour trier des entiers. Si on a besoin de trier des nombres réels, faut-il vraiment faire une autre implémentation ? L'algorithme ayant simplement besoin d'un outil pour comparer deux éléments, on devrait pouvoir utiliser le même algorithme et seulement changer l'outil pour effectuer la comparaison. Si on veut écrire le classement des participants du tour de France, on devrait toujours pouvoir utiliser le même algorithme en précisant que l'outil pour comparer est la comparaison entre les temps totaux. Ceci est particulièrement vrai pour la programmation orientée objet : on peut avoir en tête la question : est-ce que je peux écrire quelque chose de plus général, de plus abstrait ?

— laisser des possibilités/prévoir l'évolution du code dans le futur : autre exemple, on a besoin de programmer un serveur qui communique avec un client. On peut choisir une solution qui fait

exactement cela. On peut aussi penser qu'un jour on voudra communiquer avec plusieurs clients à la fois, et on peut donc choisir une solution qui permettra de faire ce changement facilement (avec de simples modifications).

De même que lorsque l'on résout des problèmes algorithmiques avec un papier et un crayon, un projet programmé dans un langage orienté objet se prépare aussi avec un papier et un crayon : différentes solutions sont possibles et le choix final fera sûrement des compromis entre performances, ré-utilisation du code et possibilités pour faire évoluer le code dans le futur. Le langage JAVA vous donne beaucoup d'outils, ce qui offre autant de choix pour construire un projet. Ce cours a pour but de passer en revue une bonne partie de ces outils en évoquant leurs possibles utilisations.

Table des matières

1 Eléments de base **1**
1 Ecrire du JAVA . . . 1
1.1 Où écrire ? Extension `.java` . . . 1
1.2 Instructions et commentaires . . . 1
1.3 Un premier programme et affichage dans la console . . . 2
2 Variables . . . 4
2.1 Nom . . . 4
2.2 Déclaration d'une variable . . . 4
2.3 Types élémentaires . . . 5
2.4 Exemples d'affectation avec valeurs . . . 5
2.5 Conversion . . . 6
3 Opérations arithmétiques . . . 6
3.1 Type d'une expression . . . 8
4 Décision . . . 8
4.1 Branchements . . . 8
4.2 Itérations . . . 10
4.3 Visibilité d'une variable dans un bloc . . . 12
5 Méthodes . . . 13
5.1 Déclaration . . . 13
5.2 une méthode particulière : la méthode `main` . . . 15
6 Une structure de données : déclarer et utiliser des tableaux . . . 16

2 Compilation, exécution, machine virtuelle **19**
1 Compilation . . . 20
2 Exécution . . . 20

3 Programmation orientée objet **21**
1 Variables d'instance . . . 22
2 Méthodes d'instance . . . 22
3 Constructeurs et création d'un objet . . . 23
4 Destruction d'un objet . . . 25
5 Manipulation de références : passage par valeur, final, égalité . . . 25
6 Variables & méthodes de classe . . . 27
6.1 Variables de classe . . . 27
6.2 Méthodes de classe . . . 28
7 Retour sur la portée : Encapsulation . . . 28

4 Héritage **31**
1 Polymorphisme 31
2 `instance of` 32
3 Les membres protégés – `protected` 33
4 Redéfinition des méthodes héritées 33
4.1 Constructeur 34
4.2 Recherche dynamique d'un membre 34
5 le mot-clé `final` 36
6 La classe `Object` 36
6.1 `toString` 37
6.2 `hashCode` 37
6.3 `equals` 37
6.4 `clone` 38
6.5 `finalize` 38
6.6 `getClass` 38
7 Classes et méthodes abstraites 38
8 Interfaces 39
8.1 Quelques détails 41
8.2 Expressions lambda 42
8.3 Références de méthode 43
9 Enumération 43
9.1 Créer et utiliser un type énuméré 43
9.2 La classe `Enum` 44
9.3 Faire plus avec un type énuméré 45
10 Information durant l'exécution 46
10.1 La classe `Class` 47

5 Exceptions **49**
1 Capturer une exception : le bloc `try` ... `catch` 50
2 Déléguer la capture d'une exception : `throws` 51
3 Créer sa propre Exception 51

6 Entré et sortie, fichiers, sauvegarde **53**
1 Lire depuis la console, afficher sur la console 54
2 Lecture/Ecriture d'un fichier 55
3 Lecture/Ecriture d'un objet : la sérialisation 57

7 Notion de types paramétrés et Collections **59**
1 Type paramétré 59
1.1 Exemple 59
1.2 Types paramétrés 62
2 Collections 62
2.1 Méthodes de l'interface Collection 63
2.2 Parcourir une collection 64
2.3 Implémentations 65
2.4 Ordre 65

8 Classes Internes **67**
1 Classes internes d'instance 67
2 Classes internes `static` 68
3 Interfaces internes ou imbriquées 68

9 Gestion des classes à l'aide de Packages (espaces de noms) **71**
1 Déclaration d'une classe 71
2 Utiliser une classe d'un espace de noms 72
3 Librairies de classes 72
4 Compilation et Exécution 73

10 Générer de la documentation : l'outil `javadoc` **75**
1 Commentaires pour les classes 76
2 Commentaires pour les méthodes 76
3 Commentaires pour les variables d'instance ou de classe 76
4 Autres commentaires 77
5 Commentaire pour les packages 77
6 Générer la documentation 77

11 Archivage de classes **79**
1 Utiliser un fichier `jar` 79

12 Annotations **81**
1 La syntaxe des annotations 81
2 Exemple d'utilisation des annotations : Tests unitaires et JUnit 82
2.1 Première solution (à bannir) : Faire une méthode `main` pour tester la classe. . . . 83
2.2 Bonne pratique : écrire une classe de tests pour chaque classe développée 83
2.3 Exemple 86
2.4 Exécution des tests 88

Chapitre 1

Eléments de base

Ce chapître s'adresse surtout à un public qui n'a jamais programmé en JAVA . Il contient un bon nombre d'inexactitudes pour permettre une compréhension assez basique du langage et nous seront plus précis dans les chapîtres qui suivent.

1 Ecrire du JAVA

1.1 Où écrire ? Extension `.java`

Le langage JAVA est un langage orienté objet. La plupart du temps, on manipulera des objets en leur faisant faire des opérations. Un *classe* definit ce qu'un objet peut faire. Ceci deviendra plus clair dans le chapître 3, mais l'important pour le moment est qu'on va écrire du code JAVA s'écrit dans un fichier texte qui possède l'extension `.java`. Par convention, le nom d'une classe est une chaîne de lettres (sans espace) qui commence toujours par une lettre *majuscule*, par exemple `MaPremiereClasse`. Cette classe sera écrite dans le fichier `MaPremiereClasse.java`.

Note: Fichiers

On a indiqué que `MaPremiereClasse.java` est un fichier dit *fichier texte*. De façon abstraite, un fichier est un espace mémoire du disque dur composé de `0` et de `1` qui encodent de l'information. Selon le type de fichier, on utilisera différentes méthodes pour décoder le fichier. Un fichier texte est un fichier dont le contenu représente seulement une suite de caractères (i.e., la suite de `0` et `1` encode une suite de caractères qui peuvent être lus et compris par un humain). Par exemple, les fichiers `.doc` ou `.xls` ne sont pas des fichiers textes, ils encodent un contenu plus riche qui sera décodé par les applications `Word` ou `Excel`. Les fichiers `.html` sont des fichiers textes mais qui seront affichés facilement par un navigateur internet.

1.2 Instructions et commentaires

JAVA est un *langage*, il possède donc une grammaire pour écrire des phrases dont la syntaxe est correcte. La grammaire de JAVA est assez simple et nous allons l'introduire de façon informelle dans la suite. Pour que le code soit compréhensible par la machine, on utilisera un programme appelé *compilateur* qui va traduire le code JAVA que vous avez écrit en un code compréhensible par la machine. Cette étape de traduction est appelée *compilation*, nous en parlerons succinctement plus tard. Lors de la compilation la première étape est toujours de vérifier si la grammaire du langage est respectée. Lors de la seconde étape, le compilateur vérifie si toutes les instructions ont un sens, par exemple si une variable utilisée a bien été déclarée, si une fonction est appelée avec le bon nombre d'arguments et les bons types, etc.

Dans un code en JAVA , on aura deux grands types de « phrase » :

— soit on a une instruction en JAVA qui doit suivre la grammaire du langage et toujours se terminer par un point-virgule ; qui correspond au point à la fin d'une phrase [1].

— soit un commentaire, i.e. un texte qui va aider à expliquer le code (par exemple, il peut jouer le rôle de titre pour une partie de code ou expliquer le but du code qui suit). Un commentaire peut décrire le code, ou bien justifier les instructions (i.e. donner un argument qui prouve que le code est correct).

Il y a deux grandes manière d'écrire des commentaires :

— commentaire sur le reste de la ligne :

```
1 // la suite est un commentaire
```

— commentaire sur plusieurs ligne : il commence par /* et se termine par */.

```
1 /* ceci est un commentaire
2  sur plusieurs
3  lignes */
```

— il exite ensuite un petit langage spécial qui sert à générer automatiquement de la documentation (des pages html) à l'aide de JAVADOC. Dans ce cas, le commentaire commence par /**, le commentaire lui même doit suivre un format particulier et le commentaire se termine par */.

1.3 Un premier programme et affichage dans la console

Le grand classique : faire afficher la chaîne `Hello World` dans la console.

```
1 package introjava;
2 
3 // The classic!
4 
5  public class HelloWorld {
6    public static void main(String[] args){
7      System.out.println('' Hello World!'');
8    }
9 }
```

Nous allons décrypter ce code de manière grossière, nous verrons en détail chaque point un peu plus tard.

La première ligne indique que la classe fait partie du `package` appelé `introjava`. Pour le moment, on indiquera simplement qu'un package permet de rassembler un ensemble de classes. Cela permet entre autre d'avoir plusieurs classes de même nom dans plusieurs `packages` différents. Le nom complet de notre classe est en fait `introjava.HelloWorld`.

En ligne 3, on peut donc voir un commentaire.

`main` est une *méthode*, c'est à dire une fonction déclarée à l'intérieur d'une classe. La ligne 3 est la déclaration de la méthode : on va y trouver son nom, si elle retourne quelque chose ou non, ses arguments. Le code exécuté par l'appel de la méthode, ce que l'on appelera le *corps* de la méthode, se trouve entre les deux accolades. Ici le corps de la méthode se limite à une seule instruction ligne 7.
`main` est une méthode particulière, c'est la première méthode appelée quand un programme est exécuté.

1. On verra aussi que la définition de classes ou de fonctions ne se terminent pas par un point virgule ;

```
~$ javac introjava/HelloWorld.java
~$ java introjava/HelloWorld
Hello World!
~$ █
```

FIGURE 1.1 – Compilation et Excécution en ligne de commande

Le mot clé `static` indique que la méthode n'opère sur aucun objet (au moment de l'appel de `main`, il n'y a pas encore d'objet). La méthode ne retourne rien, ce qui est indiqué par le mot clé `void`. Finalement, on indique la visibilité de la méthode : ici, elle est `public`, ce qui veut dire que tout autre objet pourra utiliser cette méthode. On verra qu'on pourra définir la visibilité de plusieurs éléments en JAVA .
Entre parenthèse après le nom de la méthode on trouve la liste des arguments (les paramètres d'entrée de la méthode). Ici, il n'y a qu'un seul paramètre : il a pour nom `args` et c'est un tableau contenant des `String` (on le verra plus tard `[]` signifie un tableau, et ce qui précède `[]` est le type des éléments du tableau, ici il s'agit du type `String` qui représente une chaîne de caractères).
Le corps de la méthode est donc une seule instruction : cette instruction est d'afficher un message dans l'objet `System.out`, un objet qui représente la « sortie standard » du programme (la console).

Note: Ecrire dans la console

Pour écrire sur la console de sortie (pour écrire quelque chose dans une fenêtre spéciale de l'écran), on utilise cette syntaxe

```
1 | System.out.println( <chaîne de caractères> ) ;
```

Pour l'instant cette expression semble très bizarre. On appelle la méthode `println` qui écrit dans la console le texte passé en paramètre. Cette méthode se trouve dans la classe `out` qui gère les sorties. Cette classe fait elle-même partie d'une classe nommée `System` qui gère l'entrée et la sortie standard, i.e. la saisie au clavier et l'affichage à l'écran. Nous reviendrons plus tard sur les entrées et sorties en JAVA (voir Section 6). Pour le moment, quand on utilisera ce code, cela affichera à l'écran le texte passé en paramètre.

L'exécution d'un programme va s'effectuer en deux étapes.

1. Tout d'abord, on va *compiler* le code source, c'est à dire on va traduire le code dans un langage intermédiare appelé le `byte code` et on va sauvegarder dans un fichier avec l'extension `.class`. Si la traduction se passe bien, on aura deux fichiers pour notre classe : notre fichier `HelloWorld.java` et la traduction en byte code `HelloWorld.class`.

2. Ensuite on va lancer la *machine virtuelle* qui va charger le fichier `.class` et exécuter le byte code.

Une fois compilé, le byte code peut tourner sur n'importe quelle machine virtuelle (sur un PC sous Windows, Os X, Linux, sur votre téléphone ou tablette, etc...). « write once, run anywhere ».

Si vous utilisez un environnement de programmation comme `Eclipse`, ces deux étapes sont transparentes : lorsque vous appuyez sur le bouton d'exécution, l'environnement lancera la compilation puis le lancement de la machine virtuelle. Si vous utilisez un éditeur de texte et un terminal, la compulation s'effectue avec le programme `javac` (java compilator). Le lancement de la machine virtuelle se fait à l'aide de la commande `java` (cd Figure 1.1)

2 Variables

2.1 Nom

Le nom d'une variable (il en est de même poru une méthode ou une classe), doit commencer par une lettre. Le nom peut contenir des lettres, des chiffres, ainsi que les symboles _ et $. A priori, π ou déjàVue) sont des noms valides (le compilateur saura les utiliser). Cela dit, pour échanger des codes sources (les fichiers .java), si des encodages différents sont utilisés, cela risque de poser problèmes, donc en pratique, utilisez des caractères non accentués.

2.2 Déclaration d'une variable

A chaque fois qu'on utilise une variable, par exemple toto, le compilateur JAVA doit savoir la nature de la variable, c'est ce qu'on appelle son *type*. Est-ce que toto est une valeur entière, ou bien une variable qui désigne un fichier ? Dans certains langage, les variables peuvent changer de type ou on n'a pas besoin d'explicitement déclarer le type de chaque variable. Il n'en est pas ainsi pour JAVA : en JAVA , toute variable doit être *déclarée* avant de pouvoir être utilisée, ce qui est bien normal, sinon le compilateur ne sait pas ce qu'il manipule ! Une variable ne peut pas changer de type. Pour ces raisons, on dit que le langage JAVA est *fortement typé*. Dans ce qui suit, on utilise la notation <chose> pour indiquer que dans le code il faudra un mot qui représente une chose. Par exemple <type> devra être remplacé par un type JAVA , <nom> sera remplacé par le nom d'une variable, etc. Le symbole | sera utilisé pour décrire un choix possible : par exemple
<variable> | <expression> veut dire qu'il faudra soit écrire le nom d'une variable, soit une expression JAVA complète.

Il y a plusieurs variantes pour déclarer des variables :

- *déclaration simple* : <type> <nom> ;
  elle permet de déclarer une variable en indiquant tout d'abord son type puis son nom. Avec JAVA , il *n'y a pas* de valeur par défaut, toute variable doit donc être initialisée avant utilisation ! En utilisant cette variante de déclaration, il faudra dans le code initialisée la variable, ce que le compilateur de JAVA vérifiera.
- *déclaration avec affectation* :
  <type> <nom> = <valeur dans le type> | <variable> | <expression> ;
  On peut aussi déclarer une variable et l'initialiser en une seule instruction. Pour l'initialisation, on a le choix entre donner directement une valeur (qui bien sûr doit avoir le bon type), utiliser la valeur d'une autre variable, utiliser une expression dont l'évaluation aura le bon type.
- *déclaration multiple* : <type> <nom$_1$>, <nom$_2$>, ..., <nom$_k$> ;
  on peut déclarer plusieurs variables qui ont le même type dans la même instruction.
- *déclaration multiple avec affection partielle* : <type> <nom$_1$>, <nom$_2$>= <valeur dans le type>, ..., <nom$_k$> ;
  lors d'une déclaration multiple, on peut initialiser certaines variables (pas forcément toutes).

Dans ce qui précède, le type peut être soit un type élémentaire présenté ci-dessous, soit de type complexe ou des objets. On parlera de ces objets plus tard.

2.3 Types élémentaires

JAVA fournit huit types élémentaires que peuvent prendre des variables. Lorsqu'une variable est déclarée à l'aide d'un de ces types, un espace (de taille correspondant au type) est automatiquement réservé en mémoire. Dans le tableau qui suit, on indique la taille mémoire en *bits* [2] occupée par la variable.

type élémentaire	nombre de bits	interval de valeurs
`boolean`	1	deux valeurs `true` et `false`
`byte`	8	un entier compris entre -128 et 127
`short`	16	un entier compris entre -2^{15}=-32768 et $2^{15}-1$=32767
`int`	32	un entier compris entre -$2^{31} \approx -2.1 \cdot 10^9$ et $2^{31}-1 \approx 2.1 \cdot 10^9$
`long`	64	un entier compris entre -$2^{63} \approx -9.2 \cdot 10^{18}$ et $2^{63}-1 \approx 9.2 \cdot 10^{18}$
`char`	16	caractère unicode, il y a 65536 codes
`float`	32	nombre flottant norme IEEE
`double`	64	nombre flottant norme IEEE

2.4 Exemples d'affectation avec valeurs

```
1 short population ;
2 population = 30000 ;
```

Ici, on déclare un entier court (au plus 32,767) appelé `population` et on l'initialise avec 30,000. Que se passe-t-il si on utilise l'exemple suivant ?

```
1 short population = 1000000;
```

Ici, le compilateur vous indiquera que 1,000,000 est un entier, mais pas un short et vous obtiendrez une erreur. Maintenant, quel est le problème avec l'exemple suivant ?

```
1 long nbParticules = 10000000;
```

A priori 10,000,000,000 peut bien être encodé par un `long`, mais pas par un `int`. Cependant, cette ligne n'est pas correcte. Le problème qui se pose est comment le compilateur doit-il interpréter 10,000,000,000 ? En fait, la règle pour JAVA est qu'il interprète un nombre tapé comme un `int`. Le compilateur comprend ce nombre seulement comme un entier, mais il n'y parvient pas. Pour indiquer qu'il s'agit d'un long, il faut ajouter la lettre 'L' à la fin du nombre comme suit :

```
1 long nbParticules = 10000000L;
```

de même pour les `float`s et les `double`s, on peut terminer un nombre avec `'f'` pour un `float` et avec `'d'` pour un `double`.

Pour les caractères, on doit utiliser les apostrophes comme suit (attention, un `char` correspond à un *seul* caractère, nous verrons plus tard comment faire une chaîne de caractères). JAVA utilise l'encodage `UTF-16`, un `char` est donc un code de cet encodage. Dans l'exemple suivant, on utilise la lettre 'c' qui a la valeur 99 en décimal (63 en hexadécimal), que l'on peut donc aussi écrie "\u0063". Vous pouvez donc utiliser des symboles plus exotiques (ex "\u23F0" est un réveil matin, `\uD83D \uDC19` une pieuvre).

Il y a quelques codes utiles : `''` est le retour à la ligne, `''` est une tabulation.

2. un bit est un espace mémoire qui peut prendre deux valeurs : `0` ou `1`. Il faut 8 bits pour faire un *octet*

```
1 char lettre = 'c' ;
```

Finalement, le type `boolean` contient seulement deux valeurs `true` et `false`, ces deux mots sont des mots réservés du langage.

```
1 boolean test = true ;
2 test = false ;
```

2.5 Conversion

On a dit que chaque variable avait un type et qu'une variable ne pouvait pas changer de type. Cependant, on peut utiliser une variable d'un type pour initialiser une variable d'un autre type commensurable. On déclare avec affectation une première variable et on va l'utiliser pour initialiser une seconde variable. Si les variables ont le même type, il n'y a pas de problème, mais JAVA nous permet d'utiliser des types commensurables. Par exemple, on devrait pouvoir initialiser un `double` à l'aide d'un `int`. On pourrait aussi faire l'inverse, initialiser un `int` à l'aide un `float` en faisant une opération de conversion ou *cast*. On a donc la situation suivante ou l'on affecte une variable d'un certain type et où l'on utilise cette première variable pour réaliser l'affectation d'une seconde variable :

```
1 <type1> <nom1> = <valeur1> ;
2 <type2> <nom2> = <nom1> ;
```

La *conversion* ou *cast* peut rester *implicite* si le <type$_1$> est « moins fort »que le <type$_2$> : le <type$_2$> va utiliser toute l'information du <type$_1$> sans perte. Par exemple si le <type$_1$> est un `int` et le <type$_2$> est un `double`, le double utilisera la valeur de l'entier (et la partie après la virgule est initialisé à zéro). Dans ce cas, la ligne 2 du code ci-dessous n'entrainera aucune erreur de la part du compilateur.

```
1 int valeurEntiere = 17 ;
2 double valeurFlottante = valeurEntiere ;
```

La conversion doit devenir *explicite* si le <type$_1$> est « strictement plus fort »que le <type$_2$> : il faut indiquer au compilateur d'effectuer la conversion. Pour forcer la conversion, il faut indiquer le type cible de la conversion entre paranthèses comme suit :

```
2 <type2> <nom2> = (<type2>) <nom1> ;
```

La valeur du type « plus fort »est tronquée pour pouvoir prendre le type « plus faible ». Par exemple, pour convertir un `double` ou un `float` en un `int`, JAVA tronque la partie derrière la virgule. Dans l'exemple ci-dessous, la valeur de `valeurEntiere` sera de 3.

```
1 double valeurFlottante = 3.141592 ;
2 int valeurEntiere = (int) valeurFlottante ;
```

3 Opérations arithmétiques

Les tableaux ci-dessous contiennent les différents opérateurs de JAVA . Les tableaux inclus la priorité de l'opérateur pour que les expressions ne soient pas ambigues. On peut bien sûr ajouter des parenthèses pour qu'une expression complexe soit plus lisible.

Opérateur	priorité	action	exemples
+	1	signe positif	+a ; +7
-	1	signe négatif	-a ; -(a-b) ; -7
!	1	négation logique	!(a<b) ;
++		assignement et incrément de 1	n++ ;++n ;
--		assignement et incrément de 1	n++ ; --i ;

Opérateurs unaires

Opérateur	priorité	action	exemples
*	2	multiplication	a * i
/	2	division	n/10
%	3	reste de la division entière	k%n
+	3	addition	1+2
-	3	soustraction	x-5
<	5	strictement inférieur	i<n
<=	5	inférieur ou égal	i <= n
>	5	strictement supérieur	i < n
>=	5	supérieur ou égal	i >= n
==	6	égalité	i==j
!=	6	différent	i!=j
&	7	conjonction (et logique)	(i<j) & (i<n)
|	9	disjonction (ou logique)	(i<j) | (i<n)
&&	10	conjonction optimisée	(i<j) && (i<n)
||	11	disjonction optimisée	(i<j) || (i<n)

Opérateurs binaires

L'affectation d'une variable peut être fait à l'aide d'une expression. Pour certaines expressions très utilisées (incrémente un entier d'une unité par exemple), JAVA propose des raccourcis :

— `++x` incrémente `x` de 1 puis utilise sa valeur pour l'expression
— `x++` utilise la valeur de `x` pour l'expression puis incrémente `x`
— `i += 5` est le raccourci de `i = i+5`

```
1 int i=2, j = i++;
2 i=2;
3 j= ++i;
```

Après l'exécution de la ligne 1, `i` vaut 3 et `j` vaut 2. Après l'exécution de la ligne 3, `i` et `j` valent 3.

Une erreur très classique est l'utilisation du symbole `=` à la place de `==` pour l'égalité. Pour JAVA , l'expression `i=k` a bien un sens car c'est l'affectation de la variable `i` avec le contenu de la variable `j`. Etonnement, cette expression a aussi une valeur : c'est un `int` qui est positif si l'affectation se déroule bien, et négatif sinon. Un `int` pouvant être interprété comme un boolean, JAVA fait une conversion implicite. Ainsi, une ligne de code `(i=j)` peut être interprétée comme un booléen.

Il existe un opérateur conditionnel ternaire qui affecte la valeur d'une variable en fonction d'un test :

```
1 result = someCondition ? value1 : value2;
```

Si le test (une expression booléenne) `someCondition` est vérifié, alors la variable `result` prend la valeur `value1`, sinon elle prend la valeur `value2`. Dans l'exemple ci-dessous, `absX` prend pour valeur celle de la valeur absolue de `x`.

```
1  absX = (x > 0) ? x : -x;
```

3.1 Type d'une expression

Lorsque les opérandes d'une expression ne sont pas du même type, il faut savoir quel est le type de la valeur de l'expression. La méconnaissance de ces règles entraine de nombreux bugs.

```
1  int i = 5,j;
2  double x = 5.0;
3  j=i/2;
4  j=x/2;
5  double z = i/2;
```

L'affectation de la ligne 3 est bien correcte : `i/2` est une division entre deux `int`, c'est donc un `int`, et donc, on peut se servir de la valeur pour l'affectation de l'`int j`. L'affectation de la ligne 4 est quant à elle incorrecte : `x/2` est une division entre un `double` et un `int`, c'est donc un `double`. Comme on essaye d'affecter la valeur d'un `int`, il faut utiliser le transtypage explicite. La ligne 5 est correcte, mais quelle est la valeur de `z` ? `i` est un `int`, `2` aussi, donc on a une division entre deux `int` dont le résultat est un `int` ! On a donc la division entière entre 5 et 2, dont le résultat est 2. Cette valeur est donc stockée dans le `double z`, donc `z` a pour valeur `2.0`. Si on voulait obtenir 2.5, une astuce est de forcer une division entre un entier et un double par exemple en utilisant l'expression :

```
5  double z = i/2.0;
```

Le transtypage explicite peut être vu comme un opérateur unaire qui a une priorité haute. Lors des opérations de transtypage explicite, il faut bien faire attention à la portée du transtypage, i.e. sur quelle expression porte le transtypage.

```
1  double x = 2.75;
2  int y = (int) x * 2;
3  int z = (int) (x * 2);
```

La valeur de `y` après l'exécution de la ligne 2 sera 4 car la conversion porte simplement sur `x`, et donc la multiplication s'opère entre `int`. La valeur de `z` après exécution sera 5 car la conversion porte sur le résultat de la multiplication entre un `double` et un `int`, qui est un `double`.

4 Décision

JAVA propose des structures très classiques pour effectuer certaines instructions en fonction de la valeur de certaines variables.

4.1 Branchements

Condition

La structure `if ... then ... else`

Cette structure permet de choisir d'exécuter un code différent selon une condition : si une condition est remplie, on exécute un bloc d'instructions, sinon on exécute un autre bloc d'instructions. La structure est la suivante :

```
if ( <expression booléenne> )
   <bloc d'instructions à exécuter si la condition est satisfaite>
else
   <bloc d'instruction à exécuter si la condition n'est pas satisfaite>
```

La partie « sinon ». i.e. le `else` et le bloc d'instruction qui suit est optionnel, on peut juste donc avoir une structure si une condition est remplie, alors on exécute un bloc d'instructions.

Dans la structure présentée ci-dessus, on utilise des blocs d'instructions, donc a priori après la condition, on ouvre une accolade pour le bloc d'instruction et on le ferme juste avant le `else`. Si un bloc est composé d'une seule instruction, on n'a pas besoin d'utiliser les accolades.

```
int gains,payment,encaissement;
...
// opérations qui modifient la variable gains
...
if (gains>0)
  encaissement = gains;
else
  payment = gains;
```

On peut imbriquer une autre structure de condition dans une structure existente, de cette façon, on peut utiliser plusieurs tests : si une condition est satisfaite, on fait cela, *sinon si* une autre condition est satisfaire, on fait ceci, etc.

```
int gains,payment,encaissement,investissement;
...
// opérations qui modifient la variable gains
...
if (gains<0)
  payment = gains;
else if (gains > 10) {
  encaissement = 10;
  investissement = gains-10;
}
else
  encaissement = gains;
```

On a simplement indiqué que la condition devait être une expression bouléenne. Dans l'exemple ci-dessus, elle est simple et formée d'une seule condition. Bien sûr, on peut écrire des conditions plus complexes à l'aide de conjonctions (« et » logique représenté par `&&`) et de disjonctions (« ou » logique représenté par `||`). Attention, le compilateur vérifiera seulement si la condition est une expression booléenne, pas si votre test à un sens (par exemple vous écrivez une tautologie ou un test qui sera toujours faux !).

choix multiples

Si vous voulez exécuter un bloc de code différent selon les valeurs d'une variable, vous pouvez utiliser une structure appelée `switch`. Par exemple, pensez au menu quand vous appelez une société : tapez 1 si vous voulez ceci, tapez 2 si vous voulez cela, etc. La structure se présente comme dans l'exemple

ci-dessous :

```
int choix;
...
// l'utilisateur modifie la valeur de choix
...
switch(choix) {
  case 1 :
    //instructions pour le choix 1
    ...
    break;
  case 2 :
    //instructions pour le choix 2
    ...
    break;
  default
    // instruction dans le reste des cas
    ...
}
```

On indique après le mot clé `switch` la variable sur laquelle on va faire le choix. Dans l'exemple, c'est la variable `choix`. Ensuite, chaque cas commence par le mot clé `case` suivi d'une valeur. Ici, on a deux cas selon si la variable `choix` a pour valeur 1, 2, et le cas `defaut` indique ce que l'on doit faire dans tous les autres cas. Chaque cas se termine par le mot clé `break`. L'instruction `break` permet de sortir de l'expression en cours. Ici, utilisée à l'intérieur du `switch`, elle permet de sortir du switch lorsqu'un des choix correspond au cas en cours. Si vous oubliez le `break`, plusieurs cas vont être exécutés au lieu du seul cas qui convient ! !

On peut faire un `switch` sur deux types de variables : soit sur un `int` comme dans l'exemple, soit sur un `char`. Depuis la version 7 de JAVA , on peut aussi utiliser une chaîne de caractères, que nous verrons plus tard.

4.2 Itérations

On présente maintenant trois moyens de faire des boucles.

boucle for

Une boucle « `for` »est généralement utilisée lorsque l'on sait combien de fois on va réaliser l'itération. La structure de la boucle est comme suit :

```
for (<initialisation>; <condition de poursuite>; <mise à jour des valeurs>)
  <bloc d'instructions>
```

S'il y a une seule instruction dans la boucle, on peut omettre les accolades. Il est bon d'indenter son code pour voir clairement le code qui sera itéré. Attention, il n'y a pas de point-virgule juste à la fin du `(for ...)` ! Dans la partie d'initialisation, on peut avoir plusieurs déclarations et initialisations de variables. La condition de fin est une expression booléenne (par exemple, on peut avoir des conjonctions ou des disjonctions d'expressions de tests). On peut mettre à jour plusieurs valeurs à la fois. Ci-dessous voici quelques exemples :

```
1 for ( ; ; ){
2      // some instructions
3 }
```

Q : Que se passera-t-il ?

L'exemple le plus classique est le suivant :

```
0 int n=10 ;
1 for (int i=0 ; i< n ; i++ ){
2      // some instructions
3 }
```

On rappelle que `i++` est équivalent à `i=i+1`. Si par exemple on a un tableau de taille de dimension 1 et de taille `n`, on peut utiliser une boucle comme celle-ci pour faire une opération sur chaque élément du tableau.

```
0 int n=10 ;
1 for (int i=0, j=n ; j< i ; i++ ; i-- ){
2     // some instructions
3 }
```

Dans cet exemple, on utilise deux variables pour la boucle.

On verra un peu plus tard une autre syntaxe pour les boucles `for` (Section 2.2)

boucle while

Cette structure est généralement utilisée quand on ne sait pas combien de fois on doit itérer, le coeur de la boucle est donc le test d'arrêt de la boucle. La structure se comprends comme suit : tant que la condition d'arrêt est satisfaite, on continue l'itération. La boucle s'écrit comme suit :

```
1  while(<condition de fin>)
2     <bloc d'instructions>
```

Voici un exemple qui va essayer de déterminer une approximation de la valeur de la limite de la suite convergente $u : n \to 0.75^n$:

```
1 double epsilon = 0.0000001 ;
2 double r = 0.75, u=1 ;
3  while( u - u* r > epsilon)
4      u = u * r ;
```

boucle do while

La structure de cette boucle est similaire à une boucle `while`. Au lieu de débuter par tester la condition d'arrêt avant d'effectuer une itération, on commence ici par faire une itération avant de tester la condition d'arrêt. La boucle est naturelle lorsqu'on peut exprimer le code par une phrase comme « *on fait ce bloc d'instructions tant que la condition est vérifiée* ». La structure de la boucle est comme suit :

```
1 do
2      <bloc d'instructions>
3 while(<condition de fin>) ;
```

Attention de ne pas oublier le point-virgule à la fin du `while`. On peut donc écrire l'exemple de la boucle `while` à l'aide de la boucle `do ... while ()`.

```
double epsilon = 0.0000001;
double r = 0.75, u=1;
do
    u = u * r;
while ( u - u* r > epsilon);
```

Quelle boucle choisir ?

Selon les cas une des structure est plus élégante que les autres. Si vous connaissez le nombre d'itérations nécessaires, généralement une boucle `for` est préférable. Sinon utilisez une boucle `while` ou `do ... while`.

On peut utiliser l'instruction `break` pour sortir de l'instruction courante, donc pour sortir d'une boucle `for`. Certains utilisateurs utilisent donc cette fonctionalité pour écrire une boucle `for` là où une boucle `while` serait plus adéquate. Je recommande d'utiliser une boucle `while` car la condition de fin est mise en valeur. Comme cette condition est une source d'erreurs, elle est bien plus facile à identifier. Au contraire, si quelqu'un lit un code et voit une boucle `for` sans chercher à lire exactement le bloc d'instructions, il peut penser que le code sera itéré un nombre de fois précis, alors que ce n'est pas le cas à cause du break.

Dans l'exemple suivant, on veut chercher si un élément `k` se trouve dans un tableau d'entier, et si oui, de trouver son index dans le tableau. Le code suivant est correct.

```
int[] tableau = new int[10];
// remplissage du tableau
for ( int index = 0; i< tableau.length;index++) {
  if(tableau[index] == k)
    break;
}
```

Cependant, on ne sait pas a priori combien de fois on va itérer le test « est-ce que le i^{ime} élément est égal à `k` », on peut donc utiliser une boucle `while` comme suit :

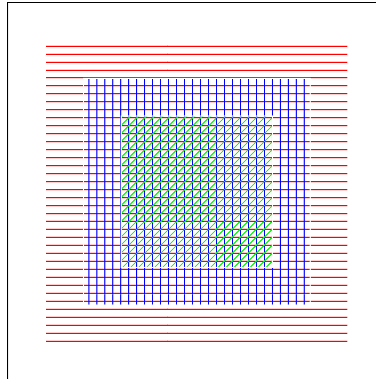
```
int[] tableau = new int[10];
// remplissage du tableau
int index = 0;
while(index < 10 && tableau[index] != k)
    index++;
```

Ces deux codes sont équivalents au niveau de la performance. Peut-être que le second est plus facile à relire.

4.3 Visibilité d'une variable dans un bloc

On peut délimiter des blocs d'instructions en les délimitant par { et }. Ainsi, on peut avoir des blocs imbriqués les uns dans les autres. On verra des utilisations particulières de blocs dans la section suivante. Mais ce qu'il est important de savoir est que les variables déclarées dans un bloc interne ne sont pas connues dans un bloc plus externe ! C'est parfois pratique lorsqu'on utilise des variables auxiliaires pour faire un calcul.

```
1  int a,b=10;
2   {
3     int d=2*b;
4     {
5       int e=b+d;
5       a=e*d;
6       {
5         int g= b+ d*e;
6       }
6     }
7  }
```



a et b sont connus partout
d est connu seulement dans la partie rouge
e est connu seulement dans la partir blue
g est connu seulement dans la partie verte

5 Méthodes

Dans un projet, on a souvent de faire les mêmes opérations. On peut donc choisir de mettre le code correspondant dans une fonction. En JAVA , on ne parle pas de fonctions, mais de *méthodes*. L'intérêt d'utiliser des fonctions est double :

— en appelant la fonction, on diminue la taille du code et le code devient plus lisible.
— si on veut modifier le code (par exemple changer l'implémentation pour améliorer les performances), il n'y a plus qu'un seul endroit à changer.

5.1 Déclaration

On parlera beaucoup plus de méthodes dans la partie de programmation orientée objet. Dans cette section, on va présenter un type de méthodes qui suit la structure suivante :

```
1  public static <type de retour> <nom>( <liste de paramètres>) {
2     corps de la méthode : suite d'instructions
3  }
```

On expliquera `public` et `static` un peu plus tard.

Une méthode peut retourner une valeur, comme une fonction en mathématique. Il faut donc indiquer le type de valeurs qui est retournée par la méthode. Cela peut être un type simple ou un type complexe comme un objet. Si la méthode ne retourne rien, on indique alors que la méthode retourne `void`.

Une méthode a bien sûr un nom et c'est ce nom qui sera utilisé lors de l'appel de la méthode. Lorsque vous le choississez, soyez créatif pour donne un nom adéquat qui donnera du sens quand vous appelerez la méthode.

Enfin, on place entre parenthèses la listes des arguments de la méthode. Ce sont les paramètres dont la méthode a besoin pour remplir sa tâche. Il n'y a pas de limite du nombre de paramètres, et on peut aussi avoir des méthodes sans paramètres (dans ce cas, il faut quand même écrire les parenthèses ouvrante et fermante). L'ordre des arguments joue un rôle important et doit être respecté lors de l'appel de la méthode.

Le nom de la méthode et la liste ordonnée des arguments constituent la *signature* de la méthode. Cette signature doit être unique, i.e., on ne peut pas avoir une autre fonction avec le même nom et la même liste d'arguments [3].

3. Ceci n'est pas complètement exact. On verra qu'avec les espaces de noms (Section 9) on peut avoir deux méthodes avec

Finalement, après cette déclaration de la méthode suit le bloc qui contient le corps de la méthode, i.e. les instructions qui doivent être exécutées. Si la méthode retourne une valeur, la méthode doit se terminer par une instruction `return` qui va renvoyer la valeur qui suit.

```
k  return <val> ;
```

Attention, une fois que cette instruction est exécutée, on va quitter la méthode et les instructions qui pourraient se trouver en dessous de l'expression `return` ne seront pas exécutées.

Dans l'exemple ci dessous, on va écrire une fonction qui va chercher l'élément maximum d'un tableau de `int`, elle va donc retourner un `int` et va prendre comme paramètres un tableau de `int`.

```
1  public static int max( int[] tableau) {
2    int m= tableau[0] ;
3    for (int i=1 ;i<tableau.length ; i++){
4      if (tableau[i] > m)
5        m = tableau[i] ;
6    }
7    return m ;
8  }
```

On peut avoir une fonction avec le même nom mais une liste d'arguments différents. Par exemple, on peut facilement faire une autre fonction `max` qui retourne le maximum d'un tableau de `double`. Cela s'appelle *surcharger une méthode*. Lors de l'exécution, JAVA choisira la méthode appropriée selon le type des arguments.

Pour appeler la fonction, il suffit simplement d'utiliser son nom et de remplir la liste des arguments en respectant bien l'ordre des arguments et leurs types. Par exemple dans l'exemple suivant, on appelle la méthode `max` que l'on vient de créer.

```
1  int tab = {7, 12, 15, 9, 11, 10, 17, 9, 13} ;
2  int m = max(tab) ;
```

En JAVA , les arguments sont toujours passés *par valeurs*. Pour les types primitifs, ce passage se fait par *copie*, c'est à dire que si on utilise une variable comme argument dans une méthode, la valeur de la variable est copiée en paramètre. La variable elle même ne sera pas affectée par l'appel de la méthode. Pour un type complexe, i.e., pour un objet ou un tableau, le passage se fait par référence, on pourra donc modifier un objet en utilisant une méthode. Dans l'exemple suivant, après l'exécution de la ligne 10, chaque entrée du tableau `tab` contient `0`, par contre la valeur de `a` sera toujours de `100`.

des signatures identiques, mais sans que cela pose des problèmes d'ambiguïté.

```
public static void zeros( int[] tableau, int a) {
   for (int i=0;i<tableau.length; i++){
     tableau[i] =0;
     a = 0;
}

public static void main(String[] arg){
   int[] tab = {10, 20, 30, 40, 50};
   int a = 100;
   zeros(tab,a);
}
```

Si vous avez déjà codé en C, vous pouvez déjà coder beaucoup de choses en utilisant la syntaxe JAVA présentée jusqu'ici. Dans le chapître suivant, on va commencer à présenter la particularité du langage JAVA : l'orientation objet.

5.2 une méthode particulière : la méthode `main`

Pour exécuter une application ou un projet, il faut appeler une méthode spéciale : la méthode `main`. Ainsi, dans l'ensemble des classes d'un projet, il faut au moins avoir défini une méthode `main` qui lancera le projet. Sa signature est fixée comme suit (si vous ne respectez pas parfaitement cette signature, JAVA ne pourra pas comprendre que cette méthode est la « véritable « méthode `main`.) :

```
public static void main(String[] args)
```

- la méthode est publique car c'est elle que l'on va appeler pour lancer l'application
- la méthode est statique. Heureusement, car sinon il faudrait déjà créer une instance, et comme `main` lance l'application, il serait difficile de créer un objet avant cela !
- la méthode ne renvoie rien, ce qui n'est pas surprenant car on se demande à qui elle renverrait une information.
- `main` est le nom de la méthode
- elle a pour seul argument un tableau de `String`. En effet, lorsqu'on lance un programme en JAVA , on tape une commande qui lance l'exécution. On peut ajouter du texte à la fin de la commande et chaque mot ajouté est inséré dans le tableau `args`. Ces mots pourront être utilisés comme option pour l'application.

Dans le code ci-dessous, l'exécution de la méthode `main` affichera Bonjour, Hello ou Hola selon le premier mot passé en option.

```java
public static void main(String[] args){
  switch(args[0]){
    case "francais" :
      System.out.println("Bonjour");
      break;
    case "english" :
      System.out.println("Hello");
      break;
  }
}
```

L'exemple ci-dessous montre deux exécutions :

```
~$ java introjava/HelloWorld francais
Bonjour
~$ java introjava/HelloWorld english
Hello
```

6 Une structure de données : déclarer et utiliser des tableaux

JAVA propose une stucture de données particulière pour faire des tableaux à n dimensions. Pour déclarer un tableau, on indique le type du tableau suivi d'autant de `[]` que le tableau a de dimension. Dans l'exemple suivant, `ligne` est un tableau à une dimension, `rectangle` un tableau a deux, et `cube` à trois.

```java
<type> [] ligne;
<type> [][] rectangle;
<type> [][][] cube;
```

Pour initialiser un tableau, il faut créé cette structure en mémoire (i.e., il faut réserver de l'espace mémoire qui contiendra le tableau). On utilise le mot clé `new` qui va effectuer cette création (on verra une utilisation similaire pour créer un nouvel objet), puis on spécifie de nouveau le type et le nombre d'éléments dans chaque dimension. Donc à la création, il faut connaître la taille maximale du tableau !

```java
<type> [] ligne = new <type>[<taille_1>];
<type> [][] rectangle = new <type>[<taille_2>][<taille_3>];
<type> [][][] cube = new <type>[<taille_4>][<taille_5>][<taille_6>];
```

Lors de la création d'un tableau, celui-ci est automatiquement initialisé avec une valeur par défaut. La valeur par défaut est

- 0 pour des tableaux contenant des types numériques (`char` inclus)
- `false` pour un tableau de `boolean`
- `null` pour un tableau contenant des objets

Pour accéder aux éléments du tableau, on utilise le nom du tableau, et les indexes dans chaque dimension. Par exemple `rectangle[3][4] + cube[1][2][5];`. On peut obtenir la taille du tableau en utilisant `.length` comme dans l'exemple suivant :

```java
int n = ligne.length;
```

On peut avoir en tête que `[]` s'applique à un type et que `<type>`, `<type>[]`, et `<type>[][]` sont des types. Donc on peut écrire le code suivant :

```java
int[][][] cube = new int[3][4][5];
int[][] rectangle = cube[2];
int n1 = cube.length;
int n2 = cube[0].length;
int n3 = cube[0][0].length;
```

Ainsi, rectangle sera un moyen pour accéder aux éléments `cube[2][i][j]` ; `n1` contiendra 3, `n2` contiendra 4 et `n3` contiendra 5.

Attention le premier élément d'un tableau a pour index `0`, et donc le dernier élément a pour index `length-1`.

Pour initialiser un tableau, on peut aussi écrire les éléments dans l'ordre entre accolades. Dans ce cas là, JAVA réserve l'espace mémoire approprié et effectue l'initialisation (on n'a pas besoin d'utiliser **new** dans ce cas là). Dans l'exemple qui suit, on initialise un tableau à une dimension et un tableau à deux dimensions.

premiers :

2	3	5	7	11	13	17	19	23

triangle :

1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

```java
int[] premiers = {2, 3, 5, 7, 11, 13, 17, 19, 23 };
int[][] triangle = {{1,1,1,1}, { 0,1,1,1}, { 0, 0, 1, 1}, {0, 0, 0, 1} };
```

Lorsqu'un nouveau tableau est créé, il occupe un espace mémoire. Dans l'exemple ci-dessous, on définit un tableau `premier` et on déclare une autre variable de type tableau d'entiers. En utilisant l'opérateur `=`, on donne un nouveau nom au même espace mémoire, mais on ne fait pas une copie du tableau. Dans l'exemple ci-dessous, il n'y a qu'un seul tableau en mémoire, donc à la fin de l'exécution, on aura `premiers[2]=4`.

```java
int[] premiers = {2,3,5,7,11]};
int[] entiers;
entiers= premiers;
entiers[2]=4;
System.out.println(premiers[2]);
```

index	valeur
0	2
1	3
2	~~5~~ 4
3	7
4	11

On verra plus tard que JAVA offre des moyens simples pour faire des opérations sur les tableaux. Nous présentons des examples maintenant, sans expliquer en détail ce qui se passe, cela sera fait Section 6.1.

— `Arrays.toString` permet d'obtenir la représentation d'un tableau en chaîne de caractères
— `Arrays.sort` trie un tableau
— `Arrays.BinarySearch` permet de chercher un élément dans un tableau

```java
int[] tab = {4, 7, 3, 1, 2};
System.out.println("tab : " + tab);
System.out.println("tab : " + Arrays.toString(tab) + "(toString)");
Arrays.sort(tab);
System.out.println("trie : " + Arrays.toString(tab));
System.out.println("index de 4 : " + Arrays.binarySearch(tab,4));
```

L'exécution du code ci-dessus produira le résultat suivant :

```
tab : [I@7852e922
tab : [4, 7, 3, 1, 2](toString)
trie : [1, 2, 3, 4, 7]
index de 4 : 3
```

Ce qui est affiché lors du premier affichage est en fait l'adresse mémoire du tableau (on peut comprendre l'ambiguité : `tab` désigne-t-il la première case mémoire du tableau ou le tableau en entier). On peut donc appeler la méthode `Arrays.toString` pour afficher les valeurs contenues dans le tableau.

Chapitre 2

Compilation, exécution, machine virtuelle

JAVA dispose d'outils pour générer et exécuter du code. En particulier, le JRE (JAVA Runtime environment) propose entre autres des bibliothèques de classes et une machine virtuelle dont nous parlerons plus bas. Le JDK (JAVA Development Kit) contient le JRE et contient en plus le compilateur et des outils pour débugger.

Ce cours traite de la programmation en JAVA . Pour pouvoir exécuter le code en JAVA , il faut tout d'abord transformer le code écrit par le programmeur, qu'on appelle le code source, en un code qui est utilisable par la machine. JAVA utilise une stratégie de machine virtuelle : le code source est traduit non dans un langage directement utilisable par la machine mais dans un language spécial appelé `bytecode`. Pour l'exécution, on utilise une machine virtuelle JAVA qui dépend de la machine d'exécution et qui va interpréter le code en `bytecode` dans le langage de la machine. Nous allons parler très succinctement de ces deux phases résumées dans le diagramme ci-dessous.

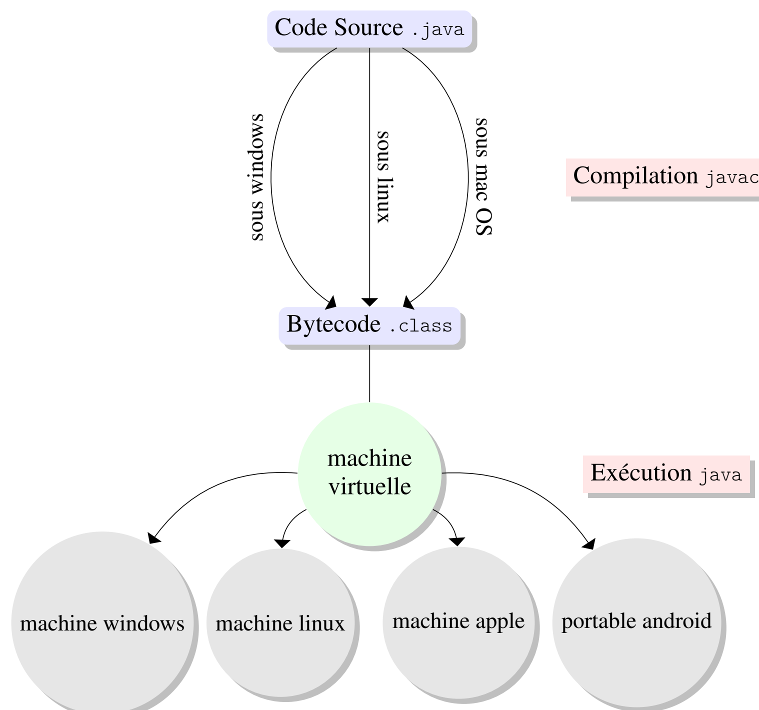


FIGURE 2.1 – Du code source à l'utilisation d'une application Java

1 Compilation

Pour créer un programme, un développeur écrit un ensemble de classes. Chaque classe <MaClasse> est enregistrée dans un fichier <MaClasse>.java : il porte le même nom que la classe et possède l'extension .java.

Ensuite, le développeur doit *compiler* l'ensemble de classes à l'aide d'un programme appelé javac. Le but du compilateur est de traduire le code écrit par le développeur en un autre langage qui pourra être exécuté par une machine. Dans le cas de JAVA , un compilateur produit un code dans le langage bytecode. Le résultat de la compilation d'une classe <MaClasse> sera un fichier nommé <MaClasse>.class, i.e., il porte aussi le nom de la classe, mais son extension est .class.

Pour simplifier, il y a deux étapes lors de la compilation :

— *analyse syntaxique* : le code est lu, on construit une répresentation du code à l'aide d'un arbre syntaxique. C'est dans cette phase qu'on vérifie la syntaxe de votre code, i.e. on vérifie la grammaire du code JAVA (pour cela, il suffit d'étudier l'arbre syntaxique créé). Le cours « Automates, Langages et application » donne les outils pour faire cette vérification.

— *analyse sémantique* : l'arbre syntaxique est analysé et traduit en bytecode. Pendant l'analyse, les références à des classes extérieures sont vérifiées (on cherche si la classe existe bien, si elle a besoin d'être compilée, etc). Le langage bycode est un langage plus simple (plus bas niveau).

2 Exécution

Une fois que les fichiers en bytecode sont écrits, on peut exécuter le programme. Dans certains langages de programmation comme le C ou le C++ la compilation génère un fichier qui sera directement exécutable par la machine : le code machine « natif » dépendra donc du système d'exploitation et de l'architecture de la machine (processeur). JAVA utilise une stratégie différente : le bytecode est un code qui peut être exécuté non par la machine directement, mais par un programme appelée une machine virtuelle. Pour chaque système opératoire (windows, linux, macintosh, etc...), JAVA fournit une machine virtuelle qui va *interpréter* le bytecode dans le langage de la machine pour l'exécuter. Cette solution possède plusieurs avantages :

— le code est *portable* : On peut écrire le code source sur une machine, compiler sur une machine d'architecture différente, et exécuter sur une troisième architecture (cd Figure 2.1). On peut utiliser les mêmes fichiers .class quelle que soit l'architecture de la machine (ordinateur, téléphone mobile, caisse enregistreuse, etc).

— la machine virtuelle permet de partager d'une manière sécurisée une machine

— le code est généralement plus compact (pas besoin d'inclure les bibliothèques comme ce serait le cas en C ou C++).

— la machine virtuelle donne l'impression que l'on dispose d'une machine entière (c'est la machine réelle qui donne du temps processeur à la machine virtuelle).

L'inconvénient majeur est que le coût en ressource de la machine virtuelle : comparé à un code exécuté directement en code natif, l'exécution d'un code JAVA sera généralement plus lente. Cependant, la machine virtuelle JAVA est très efficace et la perte en performance est assez minime.

Chapitre 3

Programmation orientée objet

JAVA est un langage de programmation orienté objet. Dans cette section, on va donc parler d'objets et de classes.

Un *objet* se définit par ses états (on peut aussi parler de ses caractéristiques) et son comportement. Par exemple, une voiture a des états : sa marque, son modèle, sa couleur, etc. Elle a aussi des comportements : elle peut accélerer, passer une vitesse, tourner, consommer du carburant, etc.

Une *classe* peut être comprise comme étant un plan ou un moule pour fabriquer des objets. Une classe va décrire les états possibles et les comportements possibles d'un objet. Les *états* d'un objet vont être représentés par des *variables* et les *comportements* d'un objet seront representés par des *méthodes*. Un *objet* est une *instance* d'une classe, i.e. à l'aide du plan ou du moule qu'est une classe, on fabrique un objet qui possède son état propre et son comportement propre. Par exemple, on peut créer une classe `Personnage` (NB : lorsqu'on nomme une classe, la convention suivie par tous est de commencer le nom de la classe par une lettre capitale). Lorsqu'on instantie la classe Personnage, on crée un objet de type `Personnage`.

La définition plus formelle d'une classe est un type abstrait caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés. Un objet ou une instance de classe possède un comportement et un état qui ne peut être modifié que par les actions du comportement.

JAVA propose déjà un bon nombre de classes. Par exemple JAVA propose une classe pour manipuler les chaînes de caractères appelée `String`. On peut donc voir cette classe comme étant un moule pour fabriquer des chaînes de caractères et ce moule décrit un ensemble de comportements qui nous aidera à manipuler ces chaînes de caractères. Ainsi, la classe `String` possède toute sorte de méthodes pour manipuler les chaînes de caractères (par exemple une méthode pour retourner le i^{ime} caractère, une méthode pour comparer la chaîne avec une autre, une méthode pour connaitre la longueur de la chaîne, pour concaténer la chaîne avec une autre, etc). Créer une variable de type `String` est synonyme d'instantier la classe `String` et de créez un objet de type `String` : une chaîne de caractères est créée et on va pouvoir la manipuler.

On va illustrer les concepts de ce chapitre à l'aide d'un exemple. Imaginons que l'on veut créer une scène d'un film d'animation. On va donc devoir manipuler les personnages du film. On veut donc commencer avec une classe `Personnage`. On note une nouvelle fois l'utilisation de la convention de nommage d'une classe : son nom commence avec une majuscule. On va donc écrire le code suivant dans le fichier `Personnage.java`

```java
public class Personnage {

}
```

le mot clé classe indique que l'on crée une classe, il est suivi par le nom de la classe et le code de la classe se trouvera entre les les accolades { et }.

1 Variables d'instance

La description de l'état d'un objet se fait par des variables qu'on appelle *variables d'instance* (elles sont aussi appelées *attributs*). Ces variables définissent les caractéristiques de l'objet. Par exemple pour la classe `Personnage`, chaque personnage doit avoir un nom.

```java
public class Personnage {
   private int age ;
   private String name ;
}
```

Le code ci-dessus implique que chaque objet ou instance de `Personnage` aura ces deux variables.

Notez ici qu'on a indiqué que ces deux variables étaient *privées*, ce qui veut dire que seulement des méthodes de la classe `Personnage` peuvent utiliser ces variables. Il y a plusieurs motivations pour cela. Une première est que vous décidez de la partie du code qui contrôle la modification d'une variable : ici, un autre objet ne peut pas directement changer la valeur de `nom`. Une autre motivation est que le programmeur garde le contrôle de la représentation interne : tant que le comportement de l'objet reste le même, vous pouvez changer tout le code.

De manière plus informelle, les états d'un objets peuvent être connus de tous ou peuvent être cachés. La voiture de James Bond n'est pas une Aston Martin grand publique, elle possède des comportements qui ne sont pas accessibles au conducteurs normaux. Ainsi, chaque variable d'instance ou de classe et chaque méthodes peuvent avoir une *portée*. Si tout le monde peut accéder à la variable ou à la méthode, la portée est publique et on la déclare en utilisant `public`. Si une variable ou une méthode ne peut être accédée que depuis l'intérieur de la classe, la portée est privée et on la déclare en utilisant le mot clé `private`. Ne pas rendre accessible à l'extérieur permet de cacher un mécanisme interne ou permet de le protéger de l'extérieur.

2 Méthodes d'instance

On va maintenant pouvoir implémenter les méthodes qui permettent de modifier l'état d'un objet. On appelle ces méthodes les *méthodes d'instance* : ces méthodes modélise le « comportement » de l'objet. Dans l'exemple ci-dessous, on a deux méthodes d'instance qui ont pour nom `getName` et `birthday`. De la même façon que pour les variables d'instance, les méthodes ont une portée, et ici, les deux méthodes sont publiques. La méthode `getName` n'a pas de paramètre et retourne une chaîne de caractères. La méthode `birthday` quant à elle retourne un entier (`int`) et n'a pas non plus de paramètres.

Il est souvent plus naturel que la portée soit publique, mais pas toujours. On fera une différence entre des méthodes qui ne modifie pas l'objet et des méthodes qui les modifient (en anglais on parle d'un « accessor » et de « mutator »). La méthode `getName` ne modifie pas l'objet alors que la méthode `birthday` modifie l'âge du personnage.

Vous ne devez laisser publique que les méthodes pertinentes pour l'utilisateur de la classe et cacher toutes les autres méthodes (surtout si elles sont dépendentes des détails d'implémentation : si l'implémentation change, vous pourrez changer ou enlever des méthodes privées sans danger pour l'utilisateur).

```
1  public class Personnage {
2
3    private String name;
4    public int age;
5
6    public String getName(){
7      return name;
8     }
9
10   public void birthday(){
11     age++;
12   }
13 }
```

Si asterix est une instance de la classe Personnage, vous pourrez donc appeler les méthodes comme suit :

```
System.out.println(asterix.getName());
asterix.birthday();
```

Si la méthode n'a pas de paramètres, il faut quand même utiliser les parenthèses (et) (cela permet de bien faire la différence avec une variable). Notez que pour le moment, nous n'avons pas de moyen pour afficher l'âge du personnage, il faudrait par exemple créer une méthode getAge.

Comme on l'a vu dans la Section 5, on peut définir plusieurs fois la même méthode avec des signatures différentes. Par exemple, on peut avoir plusieurs versions en changeant la liste des arguments. Par exemple si on veut modéliser une ellipse de plusieurs années dans notre scenario, on voudrait pouvoir ajouter d'un coup plusieurs années à notre personnage. On aura donc une méthode avec la signature suivante :

```
public int birthday(int inc)
```

Si besoin est, vous pouvez utiliser le mot clé this pour faire référence à l'objet courant. Dans l'exemple ci-dessous, le paramètre de la méthode setAge porte le même nom que la variable d'instance age. Pour éviter toute confusion, on peut dans ce cas préciser que this.age correspond à la variable d'instance et que age correspond au paramètre de la méthode.

```
14   public void setAge(int age){
15     this.age = age;
16   }
```

3 Constructeurs et création d'un objet

Une classe sert de plan ou de moule pour fabriquer un objet, ce qu'on appelle *instancier un objet*. Des méthodes spéciales, appelée des *constructeurs*, sont utilisées pour fabriquer un objet en mémoire. Ces méthodes servent donc à réserver de l'espace mémoire pour l'objet et à initialiser les variables de l'objet.

Un constructeur est une méthode qui porte le nom de la classe et qui n'a pas de type de retour. Le constructeur sans arguments, aussi appelé constructeur *par défaut* aura donc la forme suivante :

```
1  public class <nom classe> {
2    // déclaration des variable d'instances et variables de classe
3
4    // constructeur par défaut
5    public <nom classe>(){
6      // corps de la méthode
7    }
8  }
```

Comme les autres méthodes, on peut surcharger ce constructeur par défaut et avoir des constructeurs avec des signatures différentes. Pour une classe `Personnage`, on peut donc écrire :

```
1   public class Personnage {
2     public String name;
3
4     // constructeurs
5
6     // constructeur par défaut
7     public Personnage(){
8       name="unknown";
9     }
10    public Personnage(String name){
11      nom = name;
12    }
13
```

Si la valeur d'une variable d'instance n'est pas fixée explicitement dans le constructeur, elle aura automatiquement une valeur par défaut (0 pour un nombre, `false` pour un bouléen, `null` pour un objet).

Lorsque l'on déclare une variable d'instance, on peut lui donner une valeur par défaut lors de la déclaration comme dans l'exemple qui suit où la variable `name` est affectée à la chaine « `unknown` ». Cette affectation est effectuée avant l'appel du constructeur. Un constructeur pourra donc mettre à jour la valeur par défaut.

```
1  public class Personnage {
2    public String name = ''unknown'';
3
4    // constructeur
5
6    public Personnage(String name){
7      nom = name;
8    }
9
```

Vous n'êtes pas obligé d'implémenter un constructeur : dans ce cas, JAVA se charge de donner automatiquement un constructeur par défaut : celui-ci initialisera toutes les variables à leur valeur par défaut (donc soit 0, `false` ou `null` selon le type). Attention cependant, si vous avez déjà défini un constructeur avec des arguments, JAVA ne mettra pas à votre disposition un constructeur par défaut : si vous voulez

aussi avoir un constructeur par défaut, il faudra alors le définir vous-mêmes.

On a vu que pour créer une variable de type primitif (comme un `int`, un `char`, etc.), on devait déclarer le type de la variable et ensuite, éventuellement, on pouvait initialiser la variable. Pour créer une variable de type complexe, i.e. un objet, on doit aussi la déclarer. Pour créer et initialiser l'objet, il faut appeler le constructeur grâce au mot-clé **new**. Ainsi, pour créer un objet de type `Personnage` on écrit le code suivant :

```
1 Personnage asterix = new Personnage("Astérix");
```

4 Destruction d'un objet

La destruction des objets est prise en charge par JAVA avec un “garbage collector” (GC), en français litéral, c'est un mécanisme de ramassage d'ordures. Le GC détruit les objets (i.e. efface la mémoire) qui ne sont référencés par aucun autre objet. Les destructions sont asynchrones et il n'y a pas de garantie que les objets soient détruits. Si un objet à une méthode `finalize`, celle-ci est appelée lorsque l'objet est détruit. Elle peut par exemple s'assurer que des fichiers ou des connexions sont bien fermées avant la destruction de l'objet.

5 Manipulation de références : passage par valeur, final, égalité

Commençons par l'exemple suivant où on crée un personnage appelé Astérix en ligne 1. En ligne 2, on déclare une variable qui est une instance de Personnage et on l'affecte à `asterix`.

```
1 Personnage asterix = new Personnage("Astérix");
2 Personnage gaulois = asterix;
3 asterix.setAge(30);
4 gaulois.birthday();
5 System.out.println(asterix.getAge());
```

Ici, on a donc un seul objet et deux variables pour accéder à cet objet. Ces variables sont deux *références* au même objet. On peut les voir comme deux noms donnés au même objet. Si on exécute le code ci-dessus, l'affichage sera donc bien de `31`. En fait, on ne manipule jamais l'objet lui-même, on manipule toujours une référence à un objet.

Dans la Section 5, on a dit que le passage des arguments d'une méthode se faisait par valeur. C'est toujours le cas. Supposons qu'on ait la méthode suivante :

```
public static void centenaire(Personnage p){
    p = new Personnage(p.getName());
    p.setAge(100);
}
```

et supposons qu'on appelle cette méthode sur un `Personnage`.

```
1 Personnage asterix = new Personnage("Astérix");
2 asterix.setAge(30);
3 Personnage.centenaire(asterix);
4 System.out.println(asterix.getAge());
```

Lors de l'exécution de la méthode `centenaire`, on fait bien une copie de la variable `asterix` qui est une référence à l'objet Astérix : on a copié la référence seulement. Dans la méthode, cette copie change et fait référence à un nouvel objet créé dans la méthode. La méthode se termine, la variable `p` est donc détruite. L'objet créé ne sera donc référencé par personne ! Surtout, après l'exécution de la ligne 3, l'objet référencé par la variable `asterix` n'a donc pas changé ! L'affichage sera donc bien "`30`".

Par contre, changeons quelque peu notre méthode `centenaire`.

```
public static void centenaire(Personnage p){
    p.setAge(100) ;
}
```

Considérons l'appel suivant

```
Personnage.centenaire(asterix) ;
System.out.println(asterix.getAge()) ;
```

Ici, la référence `asterix` est copiée dans la variable paramètre `p`. L'objet est donc a ce moment là référencé par deux noms : `asterix` et `p`. L'objet est modifié avec l'appel de la méthode `setAge`. Après l'exécution du code, l'objet aura donc 100 comme valeur pour l'attribut age, l'affichage sera donc "`100`".

On peut déclarer une variable d'instance comme **`final`** : à la fin de l'exécution du constructeur, il ne sera alors plus possible de changer la valeur de la variable. Par exemple, on pourrait fixer comme `final` la variable `name` de la classe `Personnage`. Pour un type primitif, on ne pourra donc plus changer la valeur de la variable. Lorsqu'on a un objet, c'est de nouveau sur la référence qu'est appliqué le `final`. En effet, on pourra modifier l'objet à l'aide de méthodes qui modifient l'objet, mais on ne pourra pas modifier la référence (par exemple changer l'objet à laquelle elle est associée ou changer pour la référence `null`).

Finalement, un autre élément où la manipulation de références joue un grand rôle est sur l'égalité : il faut bien faire attention si on veut l'égalité entre deux objets ou entre deux références. Considérons l'exemple suivant :

```
1  Personnage asterix = new Personnage("Astérix") ;
2  Personnage asterixBis = new Personnage("Astérix") ;
3  Personnage hero = asterix ;
4  if (asterix == hero) ...
5  if (asterix == asterixBis) ...
```

Les variables qui sont déclarées ne contiennent pas un objet mais une *référence* vers cet objet. Le test à l'aide du symbole == permet de tester si les variables référencent le même objet.

Dans l'exemple ci-dessus, `asterix` et `hero` pointent donc sur le même objet, ainsi, le test de la ligne 4 sera positif. Par contre, la variable `asterixBis` pointe sur un objet différent, même s'il a les mêmes propriétés que l'objet `asterix`, donc, même si les objets `asterix` et `asterixBis` ont les mêmes propriétés, le test de la ligne 5 sera négatif.

Pour tester l'égalité entre les propriétés de deux objets, on doit utiliser une méthode

```
boolean equals(Object o)
```

de la classe de l'objet. Dans le cadre de l'exemple, il faudra donc ajouter à la classe `Personnage` cette méthode et définir dans quelles conditions deux personnages sont égaux.

Nous reviendrons sur le problème de l'égalité dans le chapître suivant (Section 6.3).

6 Variables & méthodes de classe

Les variables d'instance sont spécifiques à chaque instance de la classe. Mais parfois, il serait bon d'avoir une variable dont la valeur soit commune à toutes les instances d'une même classe. Par exemple, si on conçoit une classe cercle et qu'on a besoin de la valeur de π, il serait bon qu'il n'y ait qu'une valeur de π en mémoire et que toutes les instances partagent cette valeur. C'est l'idée derrière le mot clé `static`. Nous l'avons déjà rencontré dans le chapitre**??** à propos de méthodes et nous avions alors dit que ces méthodes n'opèrent sur aucun objet.

6.1 Variables de classe

Si une variable est une *variable de classe*, alors il n'y a qu'une seule variable qui sera partagée par toutes les instances de cette classe. Pour déclarer une variable de classe, on utilise le mot clé `static`. Par exemple, on va ajouter la variable de classe `cptPersonnage` :

```
1  public class Personnage {
2
3     public static int cptPersonnages ;
4     private final String name ;
5     public int age ;
6
7     public Personnage(String name){
8       this.name = name ;
9     }
```

Pour avoir accès à cette variable, il ne faut même pas avoir une instance de la classe ! En effet, pour accéder à la valeur d'une variable de classe, il suffit de concaténer le nom de la classe, le caractère point « . » et le nom de la variable de classe (comme dans l'exemple suivant) :

```
System.out.println(Personnage.cptPersonnages) ;
```

Une utilisation possible ici est de donner un numéro d'identification unique à chaque personnage. Pour ce faire, à chaque fois que l'on va créer un nouveau personnage, on va incrémenter le compteur de personnages. Ainsi chaque personnage aura un numéro unique [1].

```
1  public class Personnage {
2
3     public static int cptPersonnages ;
4     private final int id ;
5     private final String name ;
6     public int age ;
7
8     public Personnage(String name){
9       id = cptPersonnages++ ;
10      this.name = name ;
11    }
```

1. Ceci, en faisant l'hypothèse qu'il n'y a qu'un thread qui puisse créer des personnages, si on est dans une situation de programmation concurrente, il faudra remédier à ce problème.

Comme pour les variables d'instance, on peut aussi avoir des variables de classe qui sont constantes à l'aide du mot clé `final`. Un exemple intuitif est une constante mathématique comme π ou e. Un autre exemple est l'utilisation d'un générateur de nombres aléatoires : au lieu de créer plusieurs instances du générateur de nombres aléatoires de JAVA (c'est la classe `Random`), il est préférable de n'utiliser qu'une seule instance de la classe Random [2]. On pourrait donc avoir le code suivant :

```
1  public class Personnage {
2
3    public static int cptPersonnages;
4    private final int id;
5    private final String name;
6    private static final Random generator = new Random();
```

S'il y a besoin de faire des calculs pour initialiser des variables static, sachez qu'il existe de blocs d'initialisation statiques.

6.2 Méthodes de classe

De même, on a des *méthodes de classe* : ces méthodes ne modifient pas l'état interne d'un objet. On peut se servir de méthodes de classes comme des méthodes utilitaires pour manipuler des objets de la classe en question.

Un exemple de la classe `Math` est le calcul de la puissance : `Math.pow(x,a)`. Ici, il n'y a pas besoin de générer une instance de `Math`, on peut appeler la méthode de classe `pow` qui calcule x^a.

On a en fait déjà vu un exemple de méthode de classe dans le chapitre **??** dans la section sur les tableaux (Section 6). On peut utiliser des méthodes de classe de la classe `Arrays` pour faire des opérations (comme le tri, la recherche binaire, etc) sur des tableaux.

Une utilisations des méthodes de classe est ce qu'on appelle les méthodes usines (« Factory methods ») qui retournent des nouvelles instances de classes.

7 Retour sur la portée : Encapsulation

On a vu que s'il on utilise le mot-clé `public`, une variable ou une méthode était accessible par toute autre classe. Parfois, c'est très pratique, mais cela peut aussi être dangereux. Par exemple on peut changer le nom d'un personnage très facilement !

```
1  Personnage asterix = new Personnage(Astérix);
2  asterix.name = "César";
```

Parfois on veut donc éviter une modification directe par l'utilisateur des composants de l'objet. Par exemple avec l'exemple ci-dessus, on se dit qu'il faudrait que l'attribut `name` de la classe personnage devienne `private`. L'utilisateur est donc privé d'un accès direct aux (ou à certains) attributs. A la place, on va proposer des méthodes pour accéder aux variables. Ainsi, on peut penser qu'un objet est une boite noire qui rend des services à l'utilisateur : l'utilisateur a un accès direct à un certain nombre de méthodes et de variables qui vont lui rendre service. Le programmeur peut donc choisir de faire évoluer le code sans dommage pour l'application tant que les services sont identiques.

Par exemple pour un projet, on a besoin de manipuler des nombres complexes. Les méthodes vont donc être `re()`, `im()`, `rho()`, `theta()` qui donnent la partie réelle, imaginaire, le module et l'angle du

2. En effet, ce qui est aléatoire, c'est la suite des nombres

nombre complexe et des méthodes pour faire des opérations sur des complexes (addition, soustraction, multiplication, division, homothétie, rotation). Comment est représenté le nombre complexe dans l'implémentation est sans importance pour l'utilisateur. Ainsi, un programmeur peut faire une première version en utilisant une représentation cartésienne en utilisant `private double partie_re, partie_im ;` comme variables d'instances et écrire toutes les méthodes à partir de cette représentation.

Si le programmeur se rend compte que l'application fait beaucoup de calculs impliquant les rotations, la représentation polaire sera plus efficace. Il peut changer l'implémentation de la classe complexe et écrire les mêmes méthodes en utilisant la représentation polaire. Pour le reste de l'application, il n'y aura aucun changement, le changement est tout à fait transparent. Si toutefois la portée des variables d'instance `partie_re` et `partie_im` avait été laissée `public`, on aurait pu directement utiliser ces variables en dehors de la classe complexe, ce qui rendrait le passage à une implémentation en représentation polaire beaucoup plus compliqué puisqu'il faudrait modifier tout le code utilisant directement `partie_re` et `partie_im`.

Chapitre 4

Héritage

L'héritage permet à un objet d'acquérir les propriétés d'un autre objet. Si on considère notre exemple de personnage, un romain est un type de personnage, un gaulois un autre. Pour coder une classe `Gaulois`, il semble inutile de coder de nouveau tout ce qui a été codé dans la classe `Personnage`. De plus, si on veut plus tard modifier la classe `Personnage`, on ne voudrait pas qu'on ait à changer quelque chose dans la classe `Gaulois`. La solution est donc de dire que la classe `Gaulois` hérite des propriétés de la classe `Personnage`. On peut ainsi factoriser les connaissances : la classe mère (ou classe de base) est plus générale et contient les propriétés commune à toutes les classes filles (ou classe dérivée ou héritée). Les classes filles ont des propriétés plus spécifiques. On peut ainsi avoir une hierarchie de classes. De fait, tous les objets en JAVA dérivent par défaut de la classe `java.lang.Object`. Cela implique que tous les objets possèdent déjà à leur naissance un certains nombre d'attributs et de méthodes qui dérivent d'`Object`. Pour exprimer qu'une classe est une classe fille d'une autre classe, on utilise le mot-clé `extends` comme suit :

```
1 class <nom classe fille> extends <nom classe mère>
```

JAVA supporte seulement *un seul* héritage : il n'est pas possible d'hériter de plusieurs parents (un langage comme C++ permet un tel héritage multiple).

1 Polymorphisme

Par définition, un objet est une instance d'une classe. Dans l'exemple qui suit, on définit quatre classes. Tout d'abord, la classe `Personnage` est la classe la plus générale. Ensuite, on définit les classes `Gaulois` et `Romain` qui sont deux classes qui héritent de `Personnage` : un romain ou un gaulois sera un personnage particulier. Finalement, on définit une troisième classe qui hérite de `Gaulois` : il s'agit de la classe `IrreductibleGaulois`. Un irréductible gaulois est par définition un gaulois, mais c'est aussi un personnage. On voit donc bien que désormais, un objet peut avoir plusieurs types. Quand un objet peut appartenir à plusieurs types, on parle de *polymorphisme*.

```
1 public class Personnage { ... }
```

```
2 public class Gaulois extends Personnage { ... }
```

```
3 public class IrreductibleGaulois extends Gaulois { ... }
```

```
4 public class Romain extends Personnage { ... }
```

Le polymorphisme et le transtypage implicite nous permettent de manipuler des objets qui sont issus de classes différentes, mais qui partagent un même type parent : il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance).

```
1 | Personnage asterix = new Gaulois("Astérix") ;
```

On utilise alors le transtypage implicite de la classe dérivée vers la classe mère. Dans l'exemple précédent, la variable `asterix` est déclarée comme un `Personnage`. Mais cette variable fait référence à un `Gaulois`, cela est logique car un `Gaulois` est bien un `Personnage`.

A quoi cela peut bien servir ? Lorsqu'on veut utiliser une structure qui rassemble plusieurs objets, cela peut s'avérer très utile. Pour le moment, vous ne connaissez que les tableaux (on verra plus tard d'autres structures comme les listes, tas, etc). Dans l'exemple qui suit, on veut avoir un tableau contenant tous les personnages connus. On veut donc un tableau de `Personnages`, mais on ne veut pas s'interdire d'avoir des instances de sous classes de `Personnage` comme des `Gaulois`. Ici, on pourra donc utiliser une boucle pour écrire le nom de chaque personnage.

```
1 | Gaulois obelix = new Gaulois("Obélix") ;
2 | Gaulois asterix = new Gaulois("Astérix") ;
3 | Personnage cleopatre = new Personnage("Cléopâtre") ;
3 | Personnage[] distribution= new Personnage[3] ;
4 | distribution[0]= asterix ;
5 | distribution[1]= obelix ;
6 | distribution[2]= cleopatre ;
7 | for (int i=0 ;i<3 ;i++)
8 |   System.out.println(distribution[i].getName()) ;
```

Evidemment, si on utilise un type de base comme référence pour un type dérivé, on ne peut pas utiliser les méthodes spécialisées du type dérivé. En d'autres termes, si on a déclaré qu'une variable est du type `Parent`, on n'a accès qu'aux méthodes et attributs de la classe `Parent`. Même si le programmeur peut savoir qu'en fait, cet objet est de type `Enfant`, il ne peut appeler une méthode spécifique de la classe `Enfant`.

Dans notre exemple, on va supposer que la classe `Gaulois` possède une méthode `boisPotionMagique()` qui n'existe pas pour la classe `Personnage`. On ne pourra donc pas faire l'appel

`distribution[0]. boisPotionMagique().`

En effet, même si `distribution[0]` contient l'objet `asterix` qui est un `Gaulois`, `distribution[0]` est déclaré comme un `Personnage`, donc seulement les méthodes de la classe `Personnage` peuvent être utilisées.

2 `instance of`

On peut utiliser le mot clé `instance of` pour vérifier si un objet est bien une instance d'une classe ou s'il est une instance d'une sous classe. Avec le polymorphisme, un objet peut appartenir à plusieurs classes, et ainsi, il pourra retourner **`True`** à des appels différents de instance of. Par exemple, après l'exécution du code ci-dessous, on obtiendra `true`, `true` et `false` car Astérix est un `Personnage`, plus précisément, c'est bien un `Gaulois`, mais ce n'est pas un `Romain`.

```
5  ...
6  IrreductibleGaulois asterix = new IrreductibleGaulois() ;
7  System.out.println( asterix instance of Personnage) ;
8  System.out.println( asterix instance of Gaulois) ;
9  System.out.println( asterix instance of Romain) ;
```

3 Les membres protégés – `protected`

On a vu deux portées pour les variables et les méthodes : `public` et `private`. Une classe fille peut accéder aux éléments public (évidemment, comme toutes les autres classes), mais malheureusement, elle ne peut pas accéder directement aux éléments privés et les manipuler tels quels. L'idée ici est que si la classe mère possède des attributs, c'est à cette classe d'offrir tous les outils pour travailler avec ces attributs. Si le développeur a choisi de cacher des détails d'implémentation (de façon à pouvoir les changer au besoin), ces details resteront cachés pour les classes filles (et donc un changement de détails d'implémentation n'affectera pas les classes filles).

Cependant, dans certains cas, on voudrait bien avoir une portée qui permette aux classes dérivées d'avoir accès à un élément, mais pas à des classes étrangères. Il existe donc une portée appelée `protected` qui permet précisement cela : seules les classes dérivées peuvent accéder à des variables et méthodes protégées. Il vaut veiller à n'utiliser protected que lorsqu'on en a vraiment besoin.

4 Redéfinition des méthodes héritées

L'héritage permet d'ajouter des fonctionalités particulières qui ne se trouve pas dans la classe mère. Mais quant est-il pour les méthodes de la classe mère ? La classe dérivée hérite des méthodes `public` ou `protected` de sa classe mère. Ainsi, si le comportement est exactement le même que celui de la classe mère, on peut/doit omettre la ré-écriture de cette méthode. Si le comportement est différent, on peut ré-écrire la méthode avec la même signature et changer le corps de la méthode : on appelle cela la *redéfinition*.

Pour aider à faire la distinction entre une classe et sa classe mère, il existe deux références pour parcourir la hierarchie :

— *this* : est une référence sur l'instance de la classe.
— *super* : est une référence sur l'instance mère.

Dans l'exemple plus bas, la méthode `presentation()` de la classe `Gaulois` ajoute à la présentation de la classe `Personnage` le fait que le personnage est Gaulois.

Lorsqu'on redéfinit une méthode, la liste des arguments doit forcément rester la même que dans la classe mère (sinon, on effectue la création d'une nouvelle méthode et non la rédifinition d'une méthode de la classe mère). Pour cela, on peut/doit ajouter une annotation `@Override` qui permettra au compilateur de vérifier si c'est bien une rédéfinition. Par contre, il est permis de modifier le type de retour de la méthode. Par exemple, dans la classe `Romain`, on pourrait ajouter une méthode `public Romain getSuperior()`. Maintenant, si on veut faire une classe `Centurion` qui hérite de `Romain`, on peut redéfinir la méthode `getSuperior` de façon à indiquer qu'un centurion n'a pas pour supérieur un simple soldat romain, il ne peut rendre des comptes qu'à un autre centurion. On peut donc utiliser la signature `@Override public Centurion getSuperior()`.

4.1 Constructeur

Le constructeur est un cas particulier. On doit faire appel au constructeur de la classe mère à l'aide de `super`. L'appel au constructeur de la classe de base est la première instruction du constructeur de la classe dérivée. Ici, `super` indique l'appel au constructeur de la classe mère comme dans l'exemple ci-dessous.

On peut éventuellement se passer de l'appel au constructeur de la classe mère, mais seulement dans le cas où la classe mère possède un constructeur par défaut (i.e. sans arguments). Donc, que ce soit d'une manière explicite ou non, tout constructeur d'une classe fille fait *obligatoirement* appel au constructeur de la classe mère.

```
1  public class Personnage {
2
3     private String nom ;
4
5     public Personnage(String name){
6       this.nom = name ;
7     }
8
9     public String presentation(){
10      return "Je m'appelle" + name ;
11    }
12 }
```

```
1  public class Gaulois extends Personnage {
2
3     public Gaulois(String name){
4       super(name) ;
5     }
6
7     public String presentation(){
8       return super.presentation() + " je suis un gaulois";
9     }
10 }
11
12 public static void main(String[] args){
13    Gaulois asterix = new Gaulois("Astérix") ;
14    System.out.println( asterix.presentation()) ;
15 }
```

4.2 Recherche dynamique d'un membre

Si une méthode est redéfinie dans une classe fille, on peut se demander quelle méthode est effectivement appeler. Commençons par l'exemple suivant :

```
1  public class Personnage {
2      ...
3      public String presentation(){
4        return "je m'appelle "+nom ;
5      }
6  }
```

```
1  public class Gaulois extends Personnage {
2      public Gaulois(String name){ super(name) ;}
3      @Override^a
4      public String presentation(){
5        return super.presentation() + "je suis un gaulois";
6      }
7      public void frappeRomains(){
8        System.out.println("Qu'est-ce qu'on s'amuse") ;
9      }
7  }
```

a. @Override est une annotation. Elle est une indication destinée au compilateur pour lui signifier qu'il s'agit d'une redéfinition de méthode. Le compilateur vigilant vérifiera que c'est bien le cas.

```
1  public class Romain extends Personnage {
2
3      public Romain(String name){ super(name) ;}
4      @Override
5      public String presentation(){
6        return super().presentation() + " et je ne suis pas fou";
7      }
```

```
1   ...
2   public static void main(String[] args){
3     Random generator = new Random() ;
4     Personnage mystere ;
5     if (generator.nextBoolean())
6       mystere = new Gaulois("Astérix") ;
7     else
8       mystere = new Romain("Jules") ;
9     System.out.println( mystere.presentation()) ;
10  }
```

On trouve dans cet exemple une implémentation de la méthode `presentation()` dans chacune des classes. Dans la méthode `main`, on appelle la méthode `presentation()` d'un objet qui est déclaré comme un `Personnage`, mais qui est en fait soit un `Gaulois` soit un `Romain` [1]. Quelle méthode va être appelée ? Puisqu'on utilise un évènement aléatoire, on ne peut pas déduire au moment de la compilation le véritable type de l'objet `mystere`. JAVA détermine la méthode à utiliser au moment de l'exécution du programme, i.e., c'est une liasion *dynamique*. Lors de l'appel d'une méthode, JAVA remonte la hiérarchie de classes

1. Ici, on utilise la classe `Random` qui se trouve dans l'API de JAVA . Cette classe gère la création de séquence d'évènements pseudo aléatoire. Dans cet exemple, la méthode `nextBoolean` choisit de manière uniforme entre `true` et `false`.

jusqu'à trouver la méthode dont la signature correspond à l'appel. Donc dans l'exemple, JAVA va utiliser la méthode `presentation()` de la classe `Gaulois` si `mystere` est un `Gaulois`, et celle de la classe `Romain` si `mystere` est un `Romain`.

Attention ! Au moment de la compilation, on va vérifier si la méthode appliquée à un `Personnage` est bien une méthode de la classe `Personnage` ou de ses parentes. Dans l'exemple, il est vrai que la variable `mystere` fait référence à une instance de `Gaulois` ou `Romain`, néanmoins, on ne pourrait pas appeler `mystere.frappeRomains()` car la méthode `frappeRomains()` n'est définie que pour la classe `Gaulois` et non pour la classe `Personnage`.

5 le mot-clé `final`

Il est aussi possible d'indiquer qu'une méthode ne sera pas redéfinie dans une sous classe. Cela évitera à JAVA de faire la recherche de la bonne méthode à utiliser. Pour se faire, on peut utiliser le mot-clé `final`.

Ce mot-clé peut aussi être utilisé pour la définition d'une classe. Il indique alors que cette classe ne peut avoir de classes filles. Certaines classes de JAVA sont d'ailleurs déclarées comme `final`, comme par exemple la classe `String`. Le but est ici d'ordre de la sécurité : on veut être sûr que quand on manipule un `String`, on manipule bien un `String` fournie par JAVA et non une instance d'une sous classe qui pourrait faire des opérations non voulues.

Enfin, le mot-clé `final` peut aussi être utilisé pour déclarer un paramètre d'une méthode ou une variable locale d'une méthode. Cette qualification impose que le paramètre ou la variable ne sera pas modifiée (évidemment pour la variable, il faudra qu'elle soit initialisée avant la fin de l'exécution du constructeur).

6 La classe `Object`

En JAVA , que ce soit d'une façon directe ou non, toute classe hérite de la classe `Object`. Cette classe se trouve donc en haut de la hiérarchie des classes. La conséquence est qu'on pourra appeler n'importe quelle méthode définie dans la classe `Object` sur n'importe quel objet. Dans la suite, nous allons regarder plusieurs méthode de la classe `Object`. Ci-dessous, voici une partie du résumé des méthodes que l'on trouve dans la documentation JAVA .

Modifier and Type	Method Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this Object.
`int`	`hashCode()` Returns a hash code value for the object.
`String`	`toString()` Returns a string representation of the object.

6.1 toString

La méthode `toString` permet de retourner une réprésentation de l'objet avec une chaîne de caractères.

Comme c'est une méthode de la classe `Object`, on pourra donc toujours avoir une représentation d'un objet. L'implémentation dans la classe `Object` retourne le nom de la classe et le code de hachage (c'est un nombre qui identifie de maniere unique un objet, voir la section qui suit sur `hashCode()`).

Une manière un peu standard d'imprimer un objet est d'indiquer le nom de la classe suivie entre crochets de la liste des variable d'instance.

Les tableaux héritent aussi de la classe `Object`. Lorsque l'on utilise la méthode `toString`, on a une représentation quelque peu archaique, quelque chose comme ''[I@7852e922]''. Ici `I[` désigne un tableau de `int`. Pour avoir une représentation plus standard, vous pouvez utiliser la méthode `toString` de la classe Arrays du package `java.util`.

```
1  import java.util.Arrays;
2  public class TableauInt {
3
4    public static void main(String[] args){
5      int[] chiffres = 0,1,2,3,4,5,6,7,8,9;
6      System.out.println(chiffres.toString());
7      System.out.println(Arrays.toString(chiffres));
8      }
9
10   }
```

L'exécution du code ci-dessus aura pour résultat :

```
[I@7852e922
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.2 hashCode

Un code de hachage est une empreinte servant à identifier rapidement, bien qu'incomplètement, un objet. Idéalement, on voudrait un code unique, mais cela pourrait prendre un peu de temps de calculer un code unique pour une instance d'une classe. Comme on veut une méthode rapide, on va sacrifier l'unicité : si deux objets ont un code de hachage différent, la probabilité que les objets soient différents est très haute.

JAVA utilise un entier. `Object.hasCode()` dérive un code de hachage qui peut utiliser la position en mémoire, un nombre associé à l'objet, ou une combinaison.

La chose qui est importante est que deux objets égaux doivent avoir le même code de hachage. La méthode `hashCode()` doit donc être cohérente avec la méthode `equals()`.

6.3 equals

L'égalité entre deux objets fait apparaître une subtilité. La méthode `equals` implémentée dans la classe `Object` determine seulement si les références sont identiques. Pour beaucoup de cas, c'est exactement ce que l'on désire.

Par contre, parfois, on peut vouloir vérifier que deux objets ont le même contenu. Par exemple, on peut avoir deux instances de la classe `String` (qui auront donc des références différentes), mais qui seront

composés de la même chaîne de caractères. Dans ce cas, on voudra redéfinir la méthode `equals` pour comparer le contenu des objets. Attention alors dans ce cas que la méthode `hashCode()` reste cohérente avec la méthode `equals` ainsi redéfinie.

6.4 `clone`

La méthode clone est aussi subtile, et pas forcément souvent nécessaire. Ne redéfinissez pas cette méthode si vous n'avez pas une bonne raison de le faire.

Le but du clonage est de faire une nouvelle instance qui ait le même état que l'objet original. Si un des objets est modifié, l'autre devrait rester inchangé.

La méthode est déclarée comme `protected`, vous devez donc si vous voulez que les utilisateurs de votre classe puisse cloner des instances.

L'implémentation de la classe `Object` fait une copie superficielle : elle copie toutes les variables d'instance de l'objet original. Pour les variables primitives, cela est parfait. Cependant, pour les variables qui sont des objets, JAVA va simplement copier la référence.

Prenons un exemple. On a une classe `Episode` qui contient un tableau de `Personnage`. Supposons qu'on a un premier épisode avec deux personnages Astérix et Obélix. On clone ce premier episode pour en faire un second. On ajoute au second épisode le personnage d'Idéfix. Avec un clone superficiel, le tableau n'est pas dupliqué : chaque épisode réfere au même tableau en mémoire. Ainsi, les deux épisodes auront trois personnages : Astérix, Obélix et Idéfix !

6.5 `finalize`

Cette méthode sert lorsque « garbage collector » détruit l'objet (voir Section 4). Si on veut ajouter un traitement particulier lors de la destruction de l'objet, on peut ajouter le code dans cette méthode. Par exemple si l'objet est utilisé pour de la communication, l'appel de cette méthode peut servir à terminer proprement les communications.

La méthode `finalize()` de la classe `Object` ne fait en fait rien, les sous classes peuvent rédéfinir un comportement plus particulier.

6.6 `getClass`

Cette méthode permet de récupérer un objet de type `Class` qui est automatiquement généré par la machine virtuelle de JAVA . A chaque création d'objet, la machine virtuelle crée un objet de type `Class` associé qui peut permettre d'apprendre beaucoup sur l'objet initial.

7 Classes et méthodes abstraites

Prenons l'exemple de notre classe `Personnage`. Lorsque l'on va utiliser notre programme, on n'utilisera jamais d'objet de la classe `Personnage` : à chaque fois, on utilisera une classe plus précise (`Romain`, `Gaulois`, `Egyptien`, `Chien`, etc.). La classe `Personnage` est néanmoins utile comme racine de la hiérarchie et pour les méthodes que toutes ses classes filles possèderont. Pour certaines méthodes, l'implémentation dans la classe `Personnage` sera donc inutile si toutes les classes filles redéfinissent cette méthode. On voudrait déclarer ces méthodes dans `Personnage` mais sans fournir leur implémentation, et forcer à ce que l'implémentation soit fournie dans les classes filles. C'est exactement ce que propose les classes ou les méthodes dites *abstraites*.

Une méthode abstraite est une méthode qui ne possède pas de corps, seulement la signature de la méthode est donnée. Ainsi, une telle méthode doit être obligatoirement implémentée dans toutes les classes filles. L'intérêt est que l'on pourra appeler la méthode de la même façon pour tous les objets dérivés.

Une classe abstraite est une classe qui contient au moins une méthode abstraite. Une classe abstraite ne peut être instanciée directement (i.e., on ne peut pas écrire `new`), elle n'est utilisée qu'à travers sa descendance. Une classe dérivée d'une classe abstraite et qui ne redéfinit pas toutes les méthodes abstraites est elle-même abstraite.

Pour la syntaxe, les classes et les méthodes abstraites doivent être précédées du mot-clé `abstract`. Pour notre exemple, on peut faire de la classe `Personnage` une classe abstraite. Lorsqu'on veut instancier un vrai personnage, ce personnage aura un type plus précis (i.e., ce sera un gaulois, un romain, un goth, etc). Mais tous les sous-types devront implémenter un même ensemble de méthode. Ainsi quand on voudra manipuler les personnages en général, on pourra utiliser le type `Personnage`.

```
1 public abstract class Personnage {
2    public Personnage(String name) ;
3    public abstract presentation() ;
4 }
```

8 Interfaces

Commencons par un exemple. Parmi les personnages, certains sont des combattants, d'autres non. Par exemples certains gaulois vont aller combattre, mais pas tous (le druide par exemple ne combat pas), de même chez les romains, certains romains sont dans l'administration et ne combattent pas.

Une premiere solution serait de faire des sous classes de `Romain` et `Gaulois` et d'ajouter des méthodes pour combattre. Mais on sent bien qu'on va sûrement dupliquer du code. On serait tenté de faire un classe `Combattant` et on déserirerait que certains romains héritent à la fois de Combattant et de Romain. Malheureusement, JAVA permet seulement un héritage simple : on ne peut hériter que d'une seule classe. Les interfaces vont nous permettre de traiter ce probleme. Les interfaces offrent un mécanisme pour réaliser une sorte d'héritage multiple.

Une interface est un ensemble de déclarations de méthodes et de constantes qui ressemble à une classe abstraite. C'est une sorte de standard qu'une classe peut suivre : dans notre exemple, l'interface combattant va définir toutes les méthodes qu'un combattant doit posséder. Pour suivre le standard, une classe doit posséder les méthodes déclarées dans l'interface. Dans ce cas, on dit que la classe *implémente* l'interface. Une autre façon de dire est que l'on indique ce qui doit être implémenté sans donner l'implémentation.

Une classe peut suivre à la fois plusieurs standard, i.e. une classe peut implémenter plusieurs interfaces en même temps.

Une interface peut être implémentée par plusieurs classes : dans notre exemple, l'interface `Combattant` peut être implémentée par les classes `IrréductiblesGaulois`, `Pirate` ou `SoldatRomain`. Ces classes seront bien sûr différentes, mais elles ont en commun qu'elles suivent toutes le standard des combattants. On pourra donc "voir" des instances des classes `IrréductiblesGaulois`, `Pirate` ou `SoldatRomain` comme des combattants. On aura donc le droit d'écrire `Combattant c = new SoldatRomain`. L'objet `c` est en réalité un `SoldatRomain`, mais il est simplement déclaré comme un `Combattant`. On pourra donc exécuter toutes les méthodes de l'interface `Combattant`. Par contre, étant déclaré comme un `Combattant`, on ne pourra pas appeler des méthodes qui sont dans la classe `SoldatRomain` mais qui ne

sont pas dans la classe `Combattant`.

Pour déclarer une interface, on peut utiliser le modèle ci-dessous :

```
[public] interface <nom interface> [extends <nom interface 1> <nom interface 2> ... ] {
  // méthodes ou des attributs static
}
```

Donc par exemple, si on veut implémenter notre interface `Combattant`, on écrira :

```
public interface Combattant {
  public void attack(Personnage p);
  public void defense(Combattant c);
  public default void runAway(){goHome();}
}
```

Dans l'exemple, on a indiqué que l'interface `Combattant` possède 3 méthodes : `attack`, `defense`, et `runAway`. Notez qu'il n'y a pas d'implémentation pour les deux premières. Il suffit d'écrire la signature de la méthodes. Tout se passe comme si la méthode est abstraite et chaque méthode devra être implémentée par toute classe qui implémente l'interface.

Dans les anciennes versions de JAVA , les méthodes d'une interface était toujours abstraites : aucune n'avait un corps et chaque classe qui implémentait l'interface devait avoir son implémentation de chacune des méthodes de l'interface. JAVA offre désormais la possibilité d'avoir des implémentations : on pourra ainsi trouver des méthodes `static` et des méthodes par défaut (comme c'est le cas pour la méthode `runAway`). Les méthodes `static` seront par exemple des méthodes "outils" (comme pour les méthodes `static` dans des classes il n'y a pas besoin de créer un objet pour appeler ces méthodes). Les méthodes par défaut devront avoir le mot clé `default`. Dans l'exemple ci-dessous, deux méthodes devront être implémentées par les classes qui implémenteront l'interface (`attack` et `defense`) et la méthode `runAway` possède une implémentation par défaut.

Tout attribut est implicitement déclaré comme `public`, `static` et `final`. En effet, un attribut ne peut être un attribut de classe, puisqu'on se trouve dans une interface ! L'interface spécifie un comportement, et n'est pas fait pour gérer un état. C'est dans la définition d'une classe que l'on peut avoir la définition d'un état ! A la rigueur dans une interface, un attribut sera utilisé comme une constante, donc il est naturel qu'il soit `public` et `final`.

```
public class IrreductibleGaulois implements Combattant {
    ...
  public void attaque(Personnage p){
    gourdePotionMagique.bois();
    while(p.isDebout())
      coupsDePoing(p);
  }

  public void defend(Combattant c){
    esquive();
    attaque(c);
  }
}
```

L'utilisation d'interfaces s'inscrit dans la notion de polymorphisme. Un objet d'une classe qui implémente une interface possède aussi le « type »de l'interface. On peut donc avoir un tableau de `Combattants`

qui contiendra donc des objets qui implémentent l'interface `Combattant`, qu'ils soient des `SoldatRomains` ou des `IrreductiblesGaulois`. Il n'y a aucun problème pour l'appel de la méthode, puisque la signature est spécifiée dans l'interface. Lors de l'exécution de la méthode, l'implémentation utilisée sera celle qui se trouve dans les classes correspondantes ! Dans notre exemple, ceci permet d'avoir des méthodes spécialisées pour chaque type de combattants : les `Romains` n'auront évidemment pas accès à la potion magique pour combattre !

On se répète : un objet est toujours une instance d'une *classe*. Cependant, on peut utiliser une interface pour *déclarer* un objet, et dans certains cas, c'est même recommandé. Dans l'exemple ci-dessous, on a déclaré `c` comme un `Combattant`, mais en fait, on crée bien une instance de la classe `SoldatRomain`, qui se trouve être un combattant. Une fois l'objet `c` créé, ayant été déclaré comme un `Combattant`, on a accès à toutes les méthodes et attributs de l'interface `Combattant`, en particulier ici, à la méthode `attack`. De la même manière que pour l'héritage, lorsque la méthode `attack` est appelée, JAVA va exécuter la méthode qui se trouve dans "bonne" classe. Cela tombe bien car l'implémentation de la méthode `attack` ne se trouve pas dans l'interface `Combattant`, mais bien dans la classe qui implémente l'interface, donc ici dans la classe `SoldatRomain`.

```
Combattant c = new SoldatRomain() ;
c.attack() ;
```

On a dit qu'il pouvait être bon d'utiliser une interface pour déclarer des objets. Une des raisons est de travailler/raisonner à la bonne granularité. Par exemple, supposons que l'on veuille gérer une liste de `Combattants`. On verra plus tard ce que JAVA nous offre pour cela. En informatique, on a plusieur structure de donnée pour représenter une liste : par exemple les listes simplement chainées, doublement chainées, les tableaux, etc... Si on veut manipuler une liste, on pourra donc utiliser une interface `List`. Lors de la création de l'objet liste, on devra choisir une implémentation (à base de tableau par exemple). Ensuite, on pourra voir la liste comme une boite noire : les opérations vont fonctionner, les détails en interne importent alors peu ! Evidemment, la méthode `add` d'une liste chainée ne fonctionne pas du tout comme celle d'une liste à base de tableaux, mais pour l'utilisation de la liste, on veut simplement qu'un élément soit ajouté ! Evidemment, le choix de l'implémentation (list chainée ou à base de tableaux pour notre exemple) pourra avoir des répercussions sur le temps d'exécution du code par la suite.

8.1 Quelques détails

Cast et `instanceof`

Comme pour les classes, on peut utiliser le transtypage (ou cast) et l'opérateur `instanceof` pour vérifier le si un objet est bien d'un certain type.

Héritage

On peut également avoir une hierarchie d'interface. Une sous interface ajoutant donc des méthodes à la classe mère.

Conflits

Notez qu'une classe pouvant implémenter plusieurs interfaces, il peut y avoir un conflit entre méthodes. Le compilateur indiquera une erreur et il faudra lever l'ambiguité. Dans l'exemple ci-dessous, on appelle l'implémentation par défaut de l'interface `Identified`.

```java
public interface Person {
  String getName() ;
  default int getId() { return 0 ; }
  }
```

```java
public interface Identified {
  default int getId() { return Math.abs(hashCode()) ; }
  }
```

```java
public class Employee implements Person, Identified {
  public int getId() {
    return Identified.super.getId() ;
    }
  ...
  }
```

Interfaces fonctionnelles

On peut avoir des interfaces qui ne contiennent qu'une seule méthode.

8.2 Expressions lambda

Parfois, il serait intéressant de permettre qu'un paramètre soit une fonction. Par exemple, lorsqu'on trie une liste, on pourrait vouloir donner la fonction de comparaison en paramètre (si on veut trier une liste de gaulois dans l'ordre alphabétique de leurs noms ou en fonction du nombre de sangliers consommés par an). JAVA étant un langage orienté objet, *tout* est objet. Mais JAVA offre désormais une syntaxe simple pour créer des instances qui joueront le rôle de fonction.

La syntaxe ressemble fort à la syntaxe pour écrire une fonction en mathématique. Par exemple, pour la fonction carré, on écrit $x \to x^2$ en mathématique ; pour JAVA , on écrira `(Double x) -> x*x`, et voila !

La règle générale est de donner les paramètres d'entrée et leurs domaines entre parenthèse. S'il y en a plusieurs, ils sont séparés par une virgule. Ensuite les deux caractères -> indique qu'on passe maintenant à la formulation de la fonction. Ensuite, on a un bloc de code qui calcule le résultat de la fonction.

Si le type des paramètres peut être inféré par JAVA , on peut même les omettre ! Si la fonction possède un seul paramètre et qu'il peut être inféré, on peut même omettre les parenthèses !

Certaines méthodes ont comme argument une instance d'une classe qui implémente une interface dite fonctionnelle, car elle ne possède qu'une seule méthode. Dans ce cas, on peut fournir à la place une expression lambda.

Par exemple, la méthode pour trier un tableau attend une instance d'une classe qui implémente l'interface `Comparator`, et cette interface ne contient qu'une méthode `compare`. On pourra donc utiliser directement une expression pour indiquer comment comparer deux instances ! Dans l'exemple ci-dessous, on peut inférer le type des arguments `a` et `b` car ce seront des éléments du tableau `monTableau` dont le type est connu.

```java
String[] monTableau = new String[100] ;
...
Arrays.sort( monTableau, (a,b) -> a.length()-b.length()) ;
```

JAVA permet cette syntaxe, mais JAVA effectuera la transformation de cette syntaxe en un objet qui implémente l'interface `Comparator<String>`.

8.3 Références de méthode

On veut utiliser une expression lambda, mais on s'aperçoit parfois qu'il existe exactement la méthode que l'on désire. Au lieur de taper l'expression lambda, on peut se contenter de mettre la référence à la méthode. Voici quelques exemples :

```
list.removeIf( x -> x == null) ;
list.removeIf( x -> Objects.isNull()) ;
list.removeIf(Objects : :isNull) ;
```

```
list.forEach( x -> System.out.println(x)) ;
list.removeIf(System.out : :println) ;
```

On peut faire pas mal de choses technique avec tout cela, mais on se contentera de cela pour ce cours !

9 Enumération

Parfois, on a besoin d'une liste de valeurs possibles, par exemple, les tailles des vêtements sont souvent le valeurs suivantes : `XS`, `S`, `M`, `L`, et codeXL. On pourrait évidemment utiliser ces symboles et leur associer une valeur (par exemple un `int` (on verra même plus tard comment on pourrait s'assurer que la valeur associée ne puisse être changée). Mais JAVA offre un mécanisme pour directement utiliser un type énuméré.

Ce mécanisme est intéressant et utile par lui-même. On le présente aussi ici pour illustrer l'utilisation de l'héritage, du polymorphisme, et des interfaces. En effet, un type énuméré sera un nouveau type JAVA et il hérite d'une classe existente dans JAVA . De plus, cette classe implémente l'interface `Comparable`. Ce qui implique qu'une notion d'ordre par défaut existe pour tout type énuméré. On va donc pouvoir utiliser l'implémentation existente de la classe mère, ou bien redéfinir ces méthodes.

9.1 Créer et utiliser un type énuméré

Un type énuméré est une nouvelle classe, on va donc écrire le code source, comme toute classe JAVA , dans un fichier qui porte le même nom que la classe. Comme c'est un type particulier, on ne va pas utiliser le mot clé `class` mais le mot clé `enum` pour déclarer la classe. Ensuite, il suffit d'énumérer les syjmboles que l'on utilisera pour l'énumération. Ainsi, pour créer le type énuméré `Size`, on va donc écrire dans le fichier `Size.java` le code ci-dessous :

```
1 public enum Size {
2    XS,
3    S,
4    M,
5    L,
6    XL
7 }
```

Method Summary

Modifier and Type	**Method and Description**
protected Object clone()	Throws CloneNotSupportedException.
int compareTo(E o)	Compares this enum with the specified object for order.
boolean equals(Object other)	Returns true if the specified object is equal to this enum constant.
protected void finalize()	enum classes cannot have finalize methods.
Class<E> getDeclaringClass()	Returns the Class object corresponding to this enum constant's enum type.
int hashCode()	Returns a hash code for this enum constant.
String name()	Returns the name of this enum constant, exactly as declared in its enum declaration.
int ordinal()	Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).
String toString()	Returns the name of this enum constant, as contained in the declaration.
static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)	Returns the enum constant of the specified enum type with the specified name.

TABLE 4.1 – Les méthodes de la classe Enum

Après l'introduction de ce fichier, Size sera un nouveau type disponible, comme toute autre classe JAVA . Pour l'utilisation, on pourra maintenant utiliser Size comme un type dont les valeurs sont Size.XS, Size.S, Size.M, Size.L, et Size.XL.

9.2 La classe Enum

En fait, tout type énuméré hérite d'une classe existente appelée Enum, et la classe Enum implémente l'interface Comparable. Tout type énuméré possède donc les méthodes de la classe Enum, et on pourra aussi redéfinir certaines méthodes si besoin. Le tableaur̃efsummary :enum donne le résumé des méthodes de la classe Enum. On y retrouve donc les méthodes de la classe Object, ainsi que la méthode compareTo exigée par l'interface Comparable.

Par exemple, on peut utiliser la méthode values() qui retourne la liste de valeurs possibles. Ceci permet d'utiliser une boucle « pour chaque » comme suit :

```
1  public class Exemple{
2    public static void main(String[] args){
3      Size mySize = Size.M;
4      for (Size s : Size.values()){
5        if (s==mySize)
6          System.out.println("It is my size : "+s);
7        else
8          System.out.println(s + " is not my size");
9      }
10   }
11 }
```

L'exécution du code précédent donnera :

```
XS is not my size
S is not my size
It is my size : M
L is not my size
XL is not my size
```

Notez que le type énuméré est bien adapté pour faire un `switch`. A priori, on devrait pour chacun des cas utiliser le symbole de chaque élément de l'énumération, comme `Size.M` par exemple. Ici, JAVA nous aide : si on utilise un `switch` sur une variable d'un type énuméré, le compilateur JAVA va s'attendre à des valeurs du type énuméré, et on va donc pouvoir utiliser directement les symboles, par exemple écrire simplement `M` à la place de `Size.M`.

```
1  public class Exemple{
2    public static void main(String[] args){
3      Size mySize = Size.M ;
4      double price =0 ;
5      switch(mySize){
6        case S : price = 5 ; break ;
7        case M : price = 7 ; break ;
8        case L : price = 9 ; break ;
9        case XL : price = 10 ; break ;
10       }
11     System.out.println("the price is : " + price) ;
12   }
13 }
```

La classe `Enum` implémente l'interface `Comparable`. L'ordre par défaut est l'ordre dans lequel est énuméré les différents symboles. C'est un des rares cas où l'ordre de la déclaration joue un rôle très important ! La méthode `ordinal()` permet donc de savoir la position d'une valeur dans la liste de valeurs (en partant de 0). On peut bien sûr redéfinir la méthode `compareTo` si besoin est.

9.3 Faire plus avec un type énuméré

La signature du constructeur de la classe `Enum` est la suivante :

`protected Enum(String name, int ordinal).`

Ainsi, le constructeur d'un type énuméré est toujours `private` (ceci pour préserver les valeurs définies dans l'`enum`). Vous pouvez omettre le mot clé (mais mettre `public` ou `protected` provoquera une erreur de syntaxe).

Avec l'héritage, on peut aussi redéfinir le constructeur, et ainsi on peut ajouter de l'information. Voici un exemple ou cela peut s'avérer pratique : on veut utiliser comme symbole une abbréviation (comme `M`), mais on veut aussi avoir une version longue du symbole (`M` pour `Medium`).

Dans l'exemple qui suit, on fait l'inverse : les valeurs possibles du type énuméré sont `SMALL`, `MEDIUM`, `LARGE` et `EXTRA_LARGE`. A l'aide du constructeur, on ajoute une abbréviation pour la taille et une chaîne qui représente la taille française équivalente. Chaque valeur du type énuméré appelera donc une fois le constructeur. Dans la méthode `main`, on utilise donc la valeur `MEDIUM` et on demande son abbréviation et sa taille française équivalente.

```
public enum Size {
  SMALL("S", "36/38"),
  MEDIUM("M", "40/42"),
  LARGE("L","44/46"),
  EXTRA_LARGE("XL","48/50");

  private String abbreviation;
  private String eqFrance;

  Size(String abbreviation, String eqFrance) {
    this.abbreviation = abbreviation;
    this.eqFrance = eqFrance;
  }

  public String getAbbreviation() { return abbreviation; }
  public String getEqFrance(){ return eqFrance; }

  public static void main(String[] args){
    Size mySize = Size.MEDIUM;
    System.out.println("my size " + mySize.getAbbreviation()
              + " is equivalent in France to " + mySize.getEqFrance());
   }
 }
```

10 Information durant l'exécution

Nous avons vu l'instruction `instance of` qui permet de savoir si un objet est une instance d'une classe (voir Section 2). Prenons un nouvel example inspiré de la section 4.2.

```
...
public static void main(String[] args){
  Random generator = new Random();
  Personnage mystere;
  if (generator.nextBoolean())
    mystere = new Gaulois("Astérix");
  else
    mystere = new Romain("Jules");
  System.out.println( mystere instance of Gaulois);
}
```

Notez bien que dans l'example, on ne pouvait pas connaître le type de l'objet au moment de la compilation, mais seulement au moment de l'exécution du code. Nous allons voir maintenant d'autres moyen d'avoir des informations et de les utiliser pendant l'exécution.

10.1 La classe `Class`

Nous avons vu une méthode de la classe `Object` qui s'appelait `getClass` et qui retournait un objet de type `Class` (pour le moment, on va ignore les symboles <?>) qui est associé à la classe de l'objet en question. Grâce à cet objet `Class`, on va pouvoir accéder à beaucoup d'information sur la classe de l'objet, par exemple :

- on peut connaître le nom de sa classe avec la méthode `getName()`
- on peut connaître les interfaces implémentées par la classe
- on peut connaître toutes les méthodes, les variables d'instances, les constructeurs
- on peut même générer une nouvelle instance

L'exemple ci-dessous donnera toutes les méthodes de la classe `String`. On ne va pas ici donner tous les détails. Dans la boucle **`for`** ligne 8 on itère sur chaque méthode déclarée (on remarque le type `Method`), et pour chaque méthode `m`, on écrit le visibilité (« modifier ». i.e. **`public`**, **`private`**, ou **`protected`**), le type de retour, le nom de la méthode, et enfin la liste des arguments.

```
1  import java.lang.reflect.*;
2  import java.util.Arrays;
3  public class Exemple{
4     public static void main(String[] args){
5       String word = "stephane";
6       Class<?> cl = word.getClass();
7       while (cl != null) {
8         for (Method m : cl.getDeclaredMethods()) {
9           System.out.println(
10            Modifier.toString(m.getModifiers()) + " " +
11            m.getReturnType().getCanonicalName() + " " +
12            m.getName() +
13            Arrays.toString(m.getParameters()));
14        }
15        cl = cl.getSuperclass();
16      }
17    }
18 }
```

Les premieres lignes de l'exécution sont les suivantes

```
public boolean equals[java.lang.Object arg0]
public java.lang.String toString[]
public int hashCode[]
...
```

Chapitre 5

Exceptions

JAVA possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la robustesse du code (i.e. son abilité à continuer de fonctionner malgré certains problèmes). Vous avez déjà dû voir le résultat de l'exécution d'une exception quand vous avez lu des messages d'erreur.

```
1  int[] tab = new int[5];
2  tab[5]=0;
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : 5
at Personnage.main(Personnage.java :20)
```

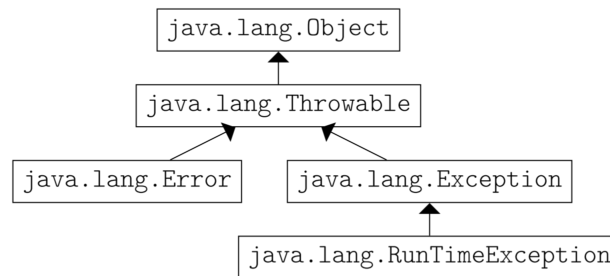
Dans l'exemple ci-dessus, le tableau est de taille 5, mais on essaie d'affecter la sixième entrée du tableau. JAVA indique la présence d'une exception. En fait, lorsqu'une erreur intervient, JAVA va instancier une classe qui correspond à l'erreur (on appelle cela lever une exception), puis va chercher à traiter l'erreur. Dans cet exemple, il s'agit d'une instance de la classe `java.lang.ArrayIndexOutOfBoundsException`. Le traitement de l'erreur est ici d'afficher le nombre 5, qui correspond à l'index du tableau que l'on cherche à accéder, mais qui n'existe pas. Ensuite, JAVA indique la méthode dans laquelle l'erreur s'est produite (ici dans la méthode `main` de la classe `Personnage`. Dans l'exemple suivant, on fait une division par 0 et on lève une exception de la classe `java.lang.ArithmeticException`, le traitement de l'exception fait simplement imprimer le message `/ by zero`.

```
1  int d=10,t1=5,t2=5;
2  System.out.println("vitesse :" + d / (t2-t1));
```

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
at Personnage.main(Personnage.java :21)
```

Dans ces exemples, l'application s'est terminée. Le but de la gestion des erreurs est de *capturer* l'exception, la traiter et ainsi parfois éviter l'arrêt du code. Certaines méthodes de JAVA sont susceptibles de produire des erreurs, et JAVA exige la prise en compte de ces erreurs potentielles (la compilation échoue si on ne le fait pas). Pour se faire, on peut faire appel à deux mécanismes : soit traiter l'exception en utilisant ce que l'on appelle un bloc `try ... catch` qui indique que faire en cas d'erreurs soit déléguer la gestion de l'erreur à la méthode appelante, dans ce cas on utilise le mot-clé `throws`. Nous allons voir cela un peu plus en détail.

La classe `Throwable` décrit l'ensemble des exceptions : à toute erreur correspond une instance d'une classe dérivée de `Throwable`. On peut avoir différents niveaux de problèmes :



La classe `Error` représente une erreur grave intervenue dans la machine virtuelle (par exemple `OutOfMemory`). Elle correspond à des erreurs matérielles ou à des erreurs de compilation. L'application JAVA s'arrête dès qu'une telle erreur est detectée, autrement dit, c'est la machine virtuelle de JAVA qui gère de telles erreurs et le programmeur ne peut pas les contrôler.

La classe `RunTimeException` regroupe aussi des exceptions qui ne peuvent être contrôlées par le programmeur.

Les autres exceptions qui dérivent de la classe `Exception` représentent des erreurs moins graves et le développeur a la possibilité de gérer de telles erreurs et ainsi éviter que l'application ne se termine, et c'est ce qui va nous occuper dans cette section.

Le traitement d'une erreur se décompose en trois étapes.

- La détection d'une erreur provoque l'instanciation d'un objet qui hérite de la classe `Exception` – c'est ce que l'on nomme lever une exception.
- la propagation (cette notion deviendra plus claire dans la section 2) : l'erreur peut être traitée « localement » ou peut être propagée récursivement à une méthode appelante grâce au mot-clé `throw`, i.e. l'erreur va être propagée à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.
- le traitement : il correspond à la capture de l'exception (utilisation des mots-clés `try` et `catch`).

1 Capturer une exception : le bloc `try ... catch`

On place dans un bloc `try` le code qui est susceptible de produire des erreurs et on capture l'exception créée avec le bloc `catch`. Lorsqu'une erreur survient dans le bloc `try`, la suite des instructions du bloc est abandonnée et le premier bloc `catch` correspondant à l'erreur est exécuté. Si on ne met rien dans le bloc `catch`, l'exception est ignorée, ce qui n'est pas une bonne pratique ! Comme plusieurs exceptions peuvent être levées par les instructions du code du bloc `try`, on peut avoir plusieurs `catch` correspondants à chacune des exceptions possibles. Attention cependant, les clauses `catch` seront testées séquentiellement. Il faut donc veiller à ne pas récuperer l'exception d'une classe et tenter de récupérer l'exception d'une classe de sa décendance. Une fois que le bloc `catch` est exécuté, l'exécution du code en dehors du bloc `try` ... `catch` se poursuit.

Dans l'exemple ci-dessous, on anticipe une erreur arithmétique (une division par 0). Dans ce cas, on imprime un message. Si une autre erreur intervient, on affiche la trace de la pile d'exécution.

```
1  int d=10,t1=5,t2=5;
2  try{
3    System.out.println("vitesse :" + d / (t2-t1));
4  }
5  catch(ArithmeticException e){
6    System.out.println(" vitesse non valide ");
7  }
5  catch(Exception e){
6    e.printStackTrace();
7  }
```

On peut faire suivre les clauses `catch` par une clause `finally` qui sera *toujours* exécutée, que l'exécution se soit passée sans problèmes ou non. Ce bloc est facultatif.

2 Déléguer la capture d'une exception : `throws`

Le bloc `try` ... `catch` permet de capturer une exception. On peut aussi choisir de déléguer la capture de l'exception à la méthode appelante (elle-même pouvant faire de même de façon récursive). On a donc le schéma suivant :

```
1  principale() {
2    ...
3    try{
4      ...
5      aux();
6      ...
7    }
8    catch(ExceptionType e){
9      // traitement exception
10   }
11 }
```

```
1  aux() throws ExceptionType {
2    ...
3    // appel à une méthode susceptible de lever une
4    // exception de type ExceptionType
5    // ou bien
6    if(condition_erreur)
7      throw new ExceptionType(args);
8  }
```

Ainsi, après la détection de l'erreur, on a donc bien besoin d'une phase de propagation pour remonter à la méthode qui va effectivement traiter l'exception. Evidemment, il faut qu'un méthode se charge de traiter l'exception et le compilateur peut vérifier si une telle méthode existe : cette méthode doit traiter précisemment le type d'exception ou bien une exception parente de l'exception qui a été levée. JAVA nous offre donc un choix de traiter finement les exceptions, ou bien de les traiter plus globalement. On peut par exemple imaginer que la méthode `main` contient un bloc `try` ... `catch(Exception e)`. Comme toutes les exceptions que le programmeur peut traiter héritent de la classe `Exception`, on aura fait un traitement général de toutes les erreurs pouvant survenir ! On a ainsi le choix d'avoir un traitement assez précis ou plutôt général de l'exception.

3 Créer sa propre Exception

Pour lever une exception, il suffit d'utiliser le mot-clé `throw` suivi d'un objet dont la classe dérive de la classe `Throwable`. Il faut alors l'indiquer dans la méthode qui contient l'instruction `throw` en ajoutant à la déclaration de la méthode le mot clé `throws` suivi du nom de la classe qui gère l'exception. Cette

indication à une valeur documentaire, mais aussi aide le compilateur à vérifier que l'exception est prise en compte à chaque fois que cette méthode est appelée.

```
1  public class PotionMagiqueException extends Exception {
2      public PotionMagiqueException(){
3        super() ;
4     }
5      public PotionMagiqueException(String s){
6        super(s) ;
7     }
8  }
```

```
1  public class GourdePotionMagique {
2     private int quantite, gorgee=2,contenance=20 ;
3     public GourdePotionMagique(){ quantite=0 ;}
4
5     public boolean bois() throws PotionMagiqueException {
6       if (quantite-gorgee <0)
7         throw new PotionMagiqueException(" pas assez de potion magique!") ;
8     }
9  }
```

Chapitre 6

Entré et sortie, fichiers, sauvegarde

On appelle entrée/sortie tout échange de données entre le programme et une source :

- entrée : au clavier, lecture d'un fichier, communication réseau
- sortie : sur la console, écriture d'un fichier, envoi sur le réseau

L'orientation objet du langage permet de regrouper des opérations similaires. JAVA utilise la notion de *flux* (*stream* en anglais) pour abstraire toutes ses opérations. Ici, le flux est un flux de données d'une source vers une destination. Le flux a donc une *direction*, on parle toujours soit d'un flux en entrée (qui pars d'une source et qui arrive dans le programme pour être traité) soit d'un flux en sortie (données traitées par le programme et qui sont envoyées à une destination). La nature de la source et de la destination peuvent être très différentes : un fichier, la console, le réseau. Le nature des données qui transitent dans le flux peut aussi varier : on distinguera principalement un flux de `bytes` (une suite de huit `0` ou `1`) ou un flux de caractères. La Figure 6.1 représente une partie de la hiérarchie de classes qui gère ces entrées/-sorties. La classe `File` contient des outils pour accéder aux informations d'un fichier, on en parlera plus tard. On peut voir deux classes abstraites qui manipulent les flux en entrée (`InputStream`) et les flux en sortie (`OutputStream`). On a représenté deux implémentations de chacune de ces classes abstraites selon la nature du flux (flux venant ou sortant d'un fichier ou d'un Objet). Finalement, on observe deux classes abstraites pour lire et écrire dans les flux (`Reader` et `Writer`). On a représenté deux implémentations qui diffèrent selon la nature du flux : un flux de `bytes` peut être traité par les classes `InputStreamReader` et `OutputStreamWriter` alors qu'un flux de caractères est traité par les classes `BufferedReader` et `Buffe-redWriter`. En résumé on obtient :

- Direction du Flux :
  - objets qui gèrent des flux d'entrée : **`in`**
    => `InputStream`, `FileInputStream`, `FileInputStream`
  - objets qui gèrent des flux de sortie : **`out`**
    => `OutputStream`, `FileOutputStream`, `FileOutputStream`
- Source du flux :
  - **fichiers** : on pourra avoir des flux vers ou à partir de fichiers
    => `FileInputStream` et `FileOutputStream`
  - **objets** : on pourra envoyer/recevoir un objet via un flux
    => `ObjectInputStream` et `ObjectOutputStream`

On peut considérer que les classes pour lire et écrire sont des outils pour transformer les flux. Un

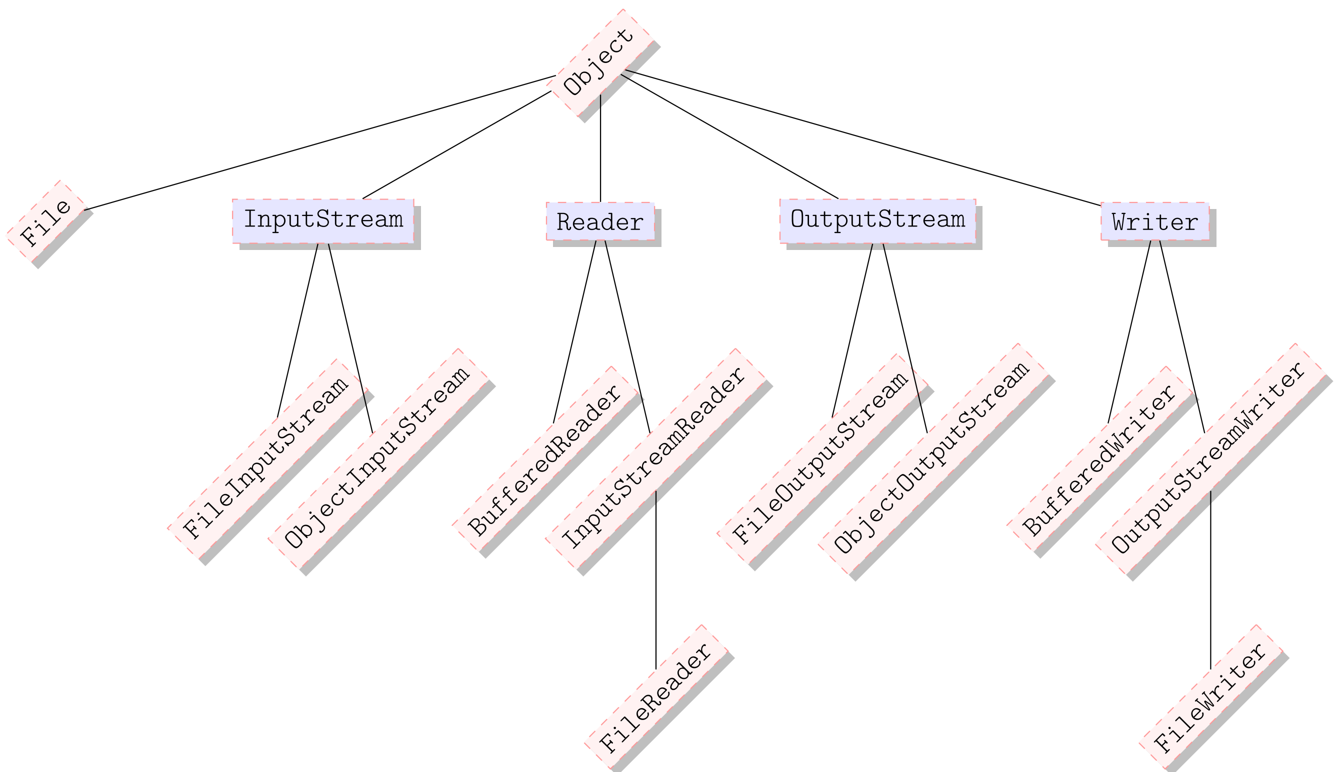


FIGURE 6.1 – Fragment de la hiérarchie de classes pour les entrées/sorties

`InputStreamReader` prend en entrée un flux de `bytes` et le transforme en un flux de caractères `char`. De manière inverse, le `OutputStreamWriter` transforme un flux de caractères en flus de `bytes`. Lorsqu'on a un flux de `char` en entrée, on peut alors utiliser la classe `BufferedReader` pour lire, via une mémoire tampon, ce qui se trouve dans le flux. De même pour la sortie, on peut écrire en sortie dans `BufferedWriter` qui utiliser une mémoire tampon et générer le flux de `char` en sortie. La Figure 6.2 résume ces étapes.

Notez bien que les opérations de lecture et d'écriture peuvent échouer pour des raisons multiples. Pour certaines, on peut avoir un contrôle (par exemple le fichier n'existe pas, on n'a pas d'accès en lecture ou écriture dans le répertoire), d'autres sont en dehors de tout contrôle (mauvais secteur du disque). Ainsi, toutes les opérations de lecture ou d'écriture sont susceptibles de lever des exceptions, qu'il faudra capturer. Il existe un hiérarchie d'exceptions dédiées aux erreurs d'entrée/sortie : ces exceptions héritent de la classe `IOException`.

1 Lire depuis la console, afficher sur la console

Pour lire, la console peut être accédée via ce que l'on appelle l'entrée « standard »`System.in`, qui est un objet de type `InputStream`. La classe `Scanner` est une classe très utile pour récupérer ce qui est tapé dans la console, en particulier récupérer et traduire automatiquement des entiers `int`, des nombres à virgule (`float` ou `double`), des chaînes de caractères (`String`).

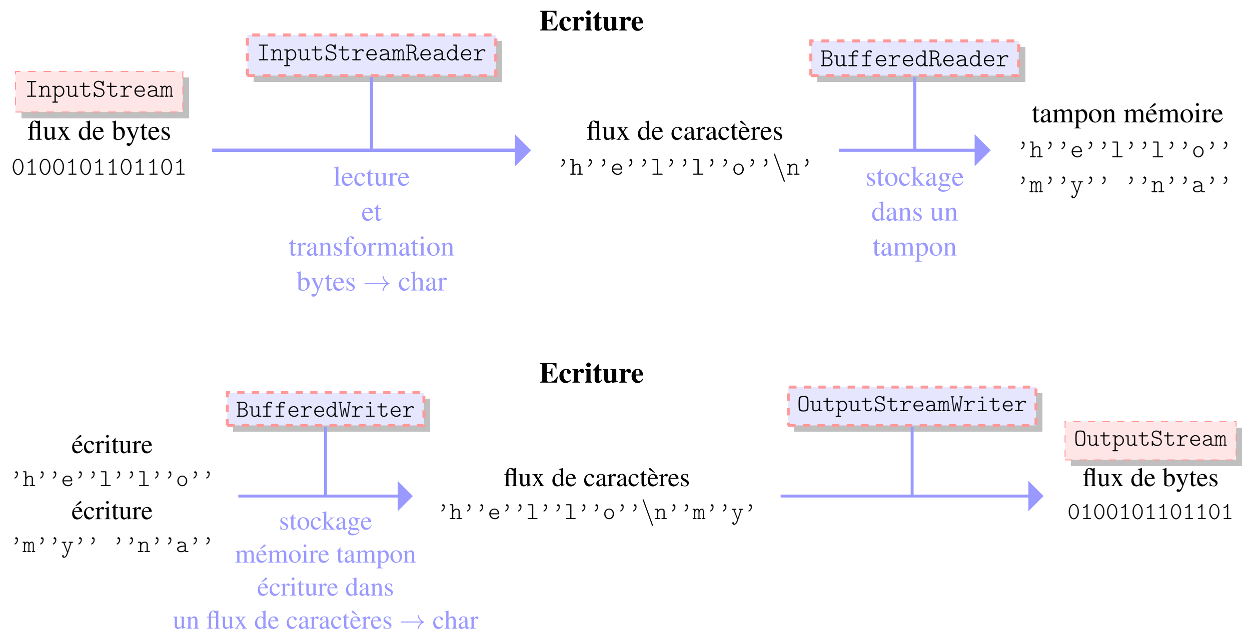


FIGURE 6.2 – Encapsulation BufferedReader(InputStreamReader) et BufferedWriter(OuputStreamWriter)

```
Scanner scan = new Scanner(System.in) ;
int n = scan.nextInt() ;
double x = scan.nextDouble() ;
String s = scan.nextLine() ;
```

Pour l'écriture, on accède la console via ce qui est appelé la « sortie standard » `System.out`, qui est de type `PrintStream` (lui-même héritant de `OutputStream`). Donc la fameuse expression `System.out.println( a_string) ;` est simplement l'écriture d'une chaîne de caractère dans un flux de `char` vers la sortie standard.

2 Lecture/Ecriture d'un fichier

La classe `File` permet d'obtenir des informations diverses sur les fichiers, on donne des exemples dans la Table 6.1. C'est aussi grâce à cette classe qu'on peut lire le contenu d'un fichier ou accéder à ce fichier pour écrire. Un fichier est une suite de `0` et `1` enregistrée sur le disque. Ainsi, pour écrire ou lire dans un fichier, JAVA utilise un flux de `bytes`. Pour écrire ou lire un fichier avec des `char`, on utilisera un objet `BufferedReader` ou `BufferedWriter`.

- En lecture : la classe `FileReader` va lire le fichier et placer le contenu dans un flux de `bytes`.
- En écriture : la classe `FileWriter` va prendre prendre un flux de bytes en entrée et écrire le flux dans le fichier

/

Dans l'exemple suivant, on va lire le premier octet d'un fichier (i.e., les 8 premières `bytes`) qui se nomme `ex.txt`. On instancie un objet de type `File` qui va accéder au fichier `ex.txt`. On va créer un flux en lecture à partir de cet objet. On aurait pu faire cela en deux étapes, mais ici, on le fait en une

— nom, chemin absolu, répertoire parent
— s'il existe un fichier d'un nom donné en paramètre
— droit : l'utilisateur a-t-il le droit de lire ou d'écrire dans le fichier
— la nature de l'objet (fichier, répertoire)
— la taille du fichier
— obtenir la liste des fichiers
— effacer un fichier
— créer un répertoire
— accéder au fichier pour le lire ou l'écrire

TABLE 6.1 – Information accessible depuis la classe `File`

seule étape en encapsulant l'instance de `File`. Ensuite, on lit le flux de `bytes` à l'aide de la méthode `read`. Evidemment, on obtient des `bytes` que l'on range dans un tableau. Une fois la lecture effectuée, on peut traiter l'information, ici, on se contente simplement d'afficher à l'écran la chaîne de caractère qui correspond à cet octet. On n'oublie bien sûr pas de fermer le flux avec la méthode `close`.

```
1 FileInputStream fis =
2                 new FileInputStream(new File(" ex.txt")) ;
3 byte[] huitLettres = new byte[8] ;
4 int nbLettreLues = fis.read(huitLettres) ;
5 for(int i=0 ;i<8 ;i++)
6    System.out.println(Byte.toString(huitLettres[i])) ;
7 fis.close() ;
```

Notez que dans ce code, on n'a pas tenu compte des exceptions :

— l'instanciation de l'objet `FileInputStream` peut lever une exception de type `FileNotFoundException`
— l'appel à la méthode `read` peut lever une exception de type `IOException`.
— la fermeture du flux avec la méthode `close` peut aussi lever une exception de typeIOException.

Il faudrait donc modifier ce code pour prendre en compte ces exceptions (soit en utilisant un bloc `try` ... `catch` soit en délégant la capture de l'exception à une méthode appelante, voir Section 5).

Dans l'exemple suivant, on affiche le contenu d'un fichier sur la console. On accède au fichier en instanciant un objet de type `File`, on utilise un flux de byte pour lire le fichier avec un objet `FileReader`, et on utilise un objet de type `BufferedReader` pour transformer le flux de `bytes` en flux de `chars`. On lit le fichier ligne par ligne en utilisant la méthode `readLine()`. De nouveau, cet exemple ne prend pas en compte les exceptions.

```
1 BufferedReader reader =
2    new BufferedReader(new FileReader(new File("ex.txt"))) ;
3 String line = reader.readLine() ;
4 while(line != null){
5    System.out.println(line) ;
6    line = reader.readLine() ;
7 }
8 reader.close() ;
```

Ci-dessous on combine les deux exemples en traitant les exceptions avec un bloc `try ... catch`.

```
try {
  FileInputStream fis = new FileInputStream(new File("test.txt")) ;
  byte[] buf = new byte[8] ;
  int nbRead = fis.read(buf) ;
  System.out.println("nb bytes read : " + nbRead) ;
  for (int i=0 ;i<8 ;i++)
    System.out.println(Byte.toString(buf[i])) ;
  fis.close() ;

  BufferedReader reader =
      new BufferedReader(new FileReader(new File("test.txt"))) ;
  String line = reader.readLine() ;
  while (line!= null){
    System.out.println(line) ;
    line = reader.readLine() ;
  }
  reader.close() ;
} catch (FileNotFoundException e) {
  e.printStackTrace() ;
}
catch(IOException e){
  e.printStackTrace() ;
}
```

3 Lecture/Ecriture d'un objet : la sérialisation

Le mécanisme de « sérialisation »permet de stocker et transmettre une instance de classe. Pour se faire, la classe en question doit implémenter l'interface `Serializable`. D'une façon surprenante, cette interface *n*'a *pas* de méthode. On peut considérer qu'elle n'a qu'un rôle de *marqueur* pour indiquer que cet objet peut être sérialisé. C'est JAVA qui se charge de traduire l'objet dans un format qui n'est pas lisible par le programmeur. Pour se faire :

— en lecture : un objet `ObjectOutputStream` traduit l'objet sérialisable en un flux de `bytes`.
— en écriture : un objet `ObjectInputStream` traduit un flux de `bytes` dans un objet.

Dans l'expemple qui suit, on va sérialiser un objet de type `IrreductibleGaulois` pour l'écrire dans un fichier et le lire par la suite. Pour la l'écriture, on utilise un `ObjectOutputStream` et on appelle la méthode `writeObject()` pour traduire l'objet. Pour la lecture, on utilise `ObjectInputStream` et sa méthod `readObject`. Notez que lors de la lecture, le compilateur ne peut savoir le type de l'objet qui est lu. C'est pour cela qu'on utilise un cast explicite, le programmeur sachant que l'objet est bien du type `IrreductibleGaulois`.

```java
IrreductibleGaulois panoramix =
        new IrreductibleGaulois("Panoramix", 1.75) ;

ObjectOutputStream oos =
        new ObjectOutputStream(
          new FileOutputStream(
            new File("panoramix.txt"))) ;

oos.writeObject(panoramix) ;
oos.close() ;

ObjectInputStream ois =
        new ObjectInputStream(
          new FileInputStream(
            new File("panoramix.txt"))) ;

IrreductibleGaulois copyPanoramix =
        (IrreductibleGaulois) ois.readObject() ;
System.out.println(copyPanoramix.nom) ;

ois.close() ;
```

Notez que dans cet exemple, le code n'est pas correct car il manque la gestion des exceptions.

Evidemment, pour qu'une classe soit « sérialisable », il serait souhaitable que tous les objets qui la composent soit sérialisable. Or, il se peut qu'un attribut ne le soit pas. Dans ce cas, on peut utiliser le mot clé `transient` pour indiquer de ne pas enregistrer cet attribut.

Chapitre 7

Notion de types paramétrés et Collections

1 Type paramétré

```
17 IrreductibleGaulois copyPanoramix = (IrreductibleGaulois) ois.readObject();
```

Dans le chapitre précédent, on a utilisé l'instruction ci-dessus : à partir d'un `ObjectInputStream`, on a récupéré un objet. Dans l'exemple, on savait qu'on allait récupérer un objet de type `IrreductibleGaulois`, mais au moment de la compilation, on ne peut pas toujours vérifier ces transformations (pour rappel, on appelle cette transformation un transtypage explicite, « cast » en anglais). Ce n'est qu'au moment de l'exécution que l'on va détecter une erreur. Pour renforcer la sécurité, JAVA offre la possibilité d'utiliser un type paramétré.

1.1 Exemple

Commençons par un exemple dans lequel on nous demande de coder une structure de données très simple : la liste chaînée. Pour commencer, on va faire une liste chaînée de `String`. On veut donc faire une liste dans laquelle chaque `String` est encapsulé dans un noeud. Le noeud contient la valeur et un lien sur le noeud suivant. Pour se faire, on écrit une classe `Noeud` et une classe `ListeChainee` comme suit

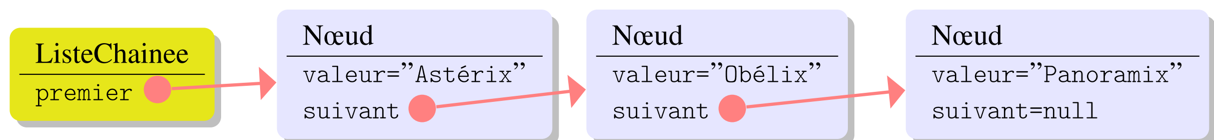
```
1  public class Noeud {
2     String valeur;
3     Noeud suivant;
4
5     public Noeud(String val){
6       valeur = val;
7     }
8
9     public void setSuivant(Noeud next){
10      suivant = next;
11    }
12 }
```

```java
public class ListChainee {
  Noeud premier ;

  public ListeChainee(){
    premier = null ;
  }

  public void add(String val){
    Noeud nouveau = new Noeud(val) ;
    if (premier == null)
      premier = nouveau ;
    else {
      Noeud dernier = premier ;
      while(dernier.suivant != null)
        dernier = dernier.suivant ;
      dernier.suivant = nouveau ;
    }
  }
}
```

Pour implémenter une liste contenant trois chaînes de caractères, par exemple {"Astérix","Obélix","Panoramix"}, en mémoire, on aura quatre objets que l'on peut représenter comme suit :



Maintenant, on voudrait faire une liste de Personnages à la place d'une liste de String.

— Une possibilité est de faire une nouvelle classe NoeudPersonnage dont les valeurs sont de type Personnage et faire de même pour une classe ListeChaineePersonnages. On devrait donc avoir une classe NoeudType par chaque type dont on voudrait faire une liste... pas très pratique. Imaginez que l'on veuille ensuite modifier la classe Noeud (par exemple ajouter une méthode, ou bien implémenter une liste avec une référence sur le noeud précédent), et il faudra faire le changement sur toutes les versions...
— Une autre possibilité est de modifier la classe Noeud et mettre Object à la place de String. On pourra mettre tous les types possibles dans un nœud, ce qui fonctionnera bien. Cependant, il faudra faire des transtypages explicites pour récupérer la valeur du nœud. Il faudra aussi faire bien attention à ce que l'on met dans la liste (on pourrait alors facilement mélanger des Strings, des Personnages, des Integers, etc... dans une même liste et il faudra un peu de travail utiliser son contenu par la suite).

La solution offerte par JAVA est la possibilité d'ajouter un type en paramètre de la classe. Ainsi, on va pouvoir avoir une seule implémentation de la classe Noeud qui va prendre en paramètre une classe, que l'on va noter E et on notera Noeud<E>. On va écrire les deux classes comme suit.

```
1  public class Noeud<E> {
2    E valeur ;
3    Noeud<E> suivant ;
4
5    public Noeud(E val){
6      valeur = val ;
7    }
8
9    public void setSuivant(Noeud<E> next){
10     suivant = next ;
11   }
12 }
```

```
1  public class ListChainee<E> {
2    Noeud<E> premier ;
3
4    public ListeChainee(){
5      premier = null ;
6    }
7
8    public void add(E val){
9      Noeud<E> nouveau = new Noeud<E>(val) ;
10     if (premier == null)
11       premier = nouveau ;
12     else {
13       Noeud<E> dernier = premier ;
14       while(dernier.suivant != null)
15         dernier = dernier.suivant ;
16       dernier.suivant = nouveau ;
17     }
18   }
19
20   public E get(int index){
21     int i=0 ;
22     Noeud<E> courant=premier ;
23     while(courant.suivant != nll && i<index){
24       i++ ;
25       courant = courant.suivant ;
26     }
27     if(index == i) // on a trouvé l'élément numéro i
28       return courant ;
28     else
29       return null ;
30     }
31   }
32 }
```

Avec cette implémentation, on va pouvoir utiliser des listes de `Personnages`, des listes de `String`, `Integer`, etc. Pour faire une liste de trois `Personnages`, on peut écrire le code suivant :

```
IrreductibleGaulois asterix = new IrreductibleGaulois("Astérix") ;
IrreductibleGaulois obelix = new IrreductibleGaulois("Obélix") ;
Gaulois Informatix = new Gaulois("Informatix") ;
ListeChainee<Gaulois> liste = new ListeChainee<Gaulois>() ;
liste.add(asterix) ;
liste.add(obelix) ;
liste.add(informatix) ;
```

Même si Astérix et Obélix sont des irréductibles gaulois, ils sont donc a fortiori des gaulois, et on peut donc manipuler une liste de gaulois. Maintenant, puisqu'il n'y a pas besoin de transtypage explicite, le compilateur peut vérifier si les types sont correctement utilisés.

Si on veut manipuler une liste de `Integer`, il faut ajouter des objets de type `Integer`, et quand on utilise un élément de la liste, on récupère un objet de type `Integer` et on doit utiliser une méthode comme `intValue()` pour obtenir la valeur de l'`Integer`. A priori, cela n'est pas très pratique, mais JAVA offre une fonctionalité qui fait automatiquement les conversions (ce qui est appelé *autoboxing*).

```
ListeChainee<Integer> maListe = new ListeChainee<Integer>() ;
//old style
maListe.add(new Integer(7)) ;
Integer sept = maListe.get(1) ;
System.out.println(sept.intValue()) ;
//new style
maListe.add(6) ;
int six = maListe.get(2) ;
```

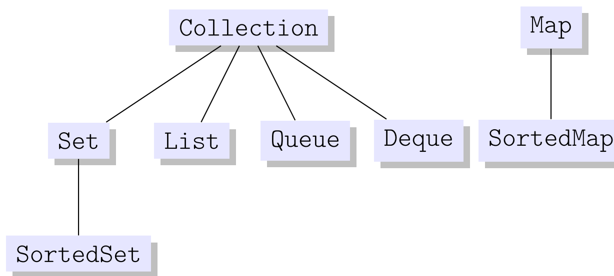
1.2 Types paramétrés

Pour ce cours, on en restera à cette introduction préliminaire des types paramétrés, car il reste beaucoup à dire. Ce sujet sera vu plus en détail lors du cours *Programmation Java Avancée*.

2 Collections

Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupent plusieurs éléments en une seule entité. Ces structures partagent un certain nombre de choses. On peut poser le même genre de questions : est-ce que la structure contient des éléments ? combien ? Certaines opérations sont similaires : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure. On peut avoir des implémentations différentes de chacune de ces structures, par exemple on a vu l'exemple de la liste chaînée, mais on pourrait avoir une implémentation basée sur un tableau, ou un liste doublement chainée (avec une référence sur l'élément précédent).

Comment peut-on manipuler toutes ces structures ? Une solution – qui devrait paraître naturelle maintenant – est d'utiliser une hiérarchie d'interfaces, et c'est ce que JAVA propose.



- `Collection` : Tout en haut de la hiérarchie se trouve l'interface `Collection`. C'est le plus petit dénominateur commun, une collection rassemble simplement un nombre d'éléments. Certains types de collections autoriseront des doublons, pas d'autres, certains types sont ordonnés. On retrouve dans cette interface des méthodes de base pour parcourir, ajouter, enlever des éléments.
- `Set` : cette interface représente un *ensemble* au sens mathématique, et donc, ce type de collection n'admet *aucun* doublon.
- `List` : cette interface représente une *séquence* d'éléments. Ainsi, l'ordre dans lequel on ajoute ou on enlève des éléments est important. On peut avoir des doublons dans la séquence.
- `Queue` : cette interface représente une *file d'attente*. Dans une file d'attente, l'élément qui est en tête est particulièrement important. En fait si important qu'on peut avoir l'image suivante d'une file d'attente : il y l'élément en tête et il y a les éléments qui suivent, dont on ne préocuppe pas. On a généralement deux types de file d'attente : celle ou le premier entré est en tête de file et celui ou le dernier entré est celui en tête de file. L'ordre dans lequel les éléments sont ajoutés ou enlevés est important et il peut y avoir des doublons.
- `Deque` : cette interface ressemble aux files d'attente, mais les éléments importants sont les éléments en tête et en queue.
- `Map` : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
- `SortedSet` est la version ordonnée d'un ensemble
- `SortedMap` est la version ordonnée d'une relation binaire ou les *clés* sont ordonnées.

On introduira de la notion d'ordre entre des objets plus tard dans ce document, mais elle sera vu pendant le cours *Programmation Java Avancée*.

Notez que ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de `Gaulois`, de `Integer`, de `String`, etc...

2.1 Méthodes de l'interface Collection

Toutes les classes qui implémentent l'interface `Collection` devront redéfinir un certain nombre de méthodes parmis lesquelles on retrouve des méthodes pour l'ajout, le retrait d'éléments, des méthodes pour savoir si un ou des éléments sont présents dans la collection, une méthode (`size`) pour connaitre le nombre d'éléments contenus dans la collection. On ne va pas expliquer en détail chaque méthode, mais vous trouverez ci-dessous une liste (non exhaustive) des méthodes de cette interface. Notez la présence de certaines méthodes génériques. On va expliquer dans la section suivante l'utilité de la méthode `iterator()`.

- `boolean add(E e)`
- `void clear()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`
- `boolean isEmpty()`

- `Iterator<E> iterator()`
- `boolean remove(Object o)`
- `int size()`
- `Object[] toArray()`

2.2 Parcourir une collection

Il y a deux moyens de parcourir une collection : soit en utilisant une boucle « pour chaque élément », qui offre une version différente de la boucle `for` vue précédemment, soit en utilisant un objet appelé `iterator`.

En utilisant la généricité, le compilateur va connaître le type des éléments qui sont contenus dans la collection. Une collection étant un ensemble d'éléments, JAVA propose une façon simple d'accéder à chacun des éléments. Dans l'exemple ci-dessous, on a une collection `maCollection` qui contient des objets de type `E`. On va accéder à chaque élément de la collection `maCollection` en utilisant le mot-clé `for`, chaque élément sera stocké dans une variable `<nom>` de type `E` (évidemment).

```
1 Collection<E> maCollection;
2 ...
3 for (E <nom> : maCollection)
4     // block d'instructions
```

Par exemple, dans l'exemple suivant, on parcours une liste de `Personnages` et on appelle la méthode `presentation()`.

```
1 LinkedList<Personnage> villageois = new LinkedList<Personnage>();
2 villageois.add(new Gaulois("Ordralfabétix"));
3 villageois.add(new Gaulois("Abraracourcix"));
4 villageois.add(new Gaulois("Assurancetourix"));
5 villageois.add(new Gaulois("Cétautomatix"));
6 for (Personnage p : villageois)
7     p.presentation();
8 
```

Une autre solution est d'utiliser un objet dédié au parcourt d'éléments dans une collection : un objet qui implémente l'interface `Iterator`. Pour obtenir cet objet, on peut appeler la méthode `iterator()` qui se trouve dans l'interface `Collection`. L'interface `Iterator`, contient les deux méthodes suivantes :

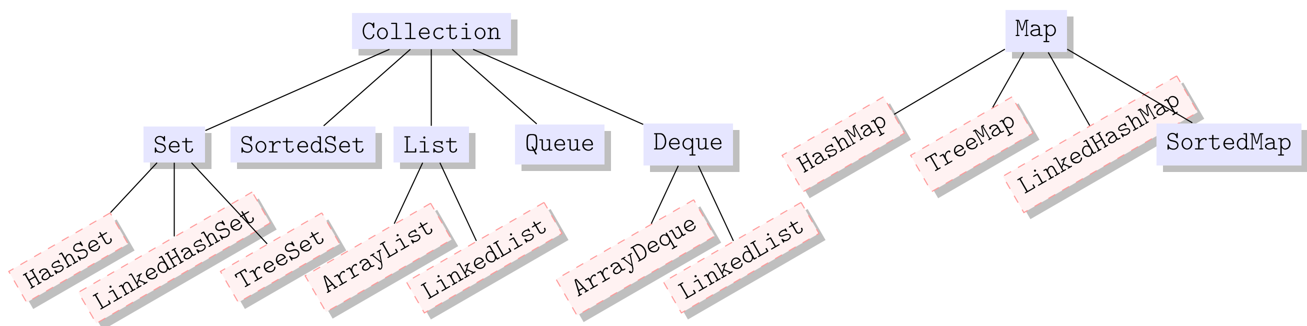
- `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
- `next()` donne accès à l'élément suivant, et `remove` permet d'enlever l'élément de la collection.

```
1 public interface Iterator<E> {
2     boolean hasNext();
3     E next();
4     void remove(); //optional
5 }
```

En fait, l'utilisation d'une boucle « pour chaque »masque l'utilisation d'un `Iterator`. En général, l'utilisation de la boucle « pour chaque »est la plus simple. Vous devez utiliser un `Iterator` lorsque vous voulez enlever un élément ou lorsque vous voulez parcourir plusieurs collections en parallèle.

2.3 Implémentations

Pour chacune des interfaces, il existe plusieurs *implémentations*. Le diagramme ci-dessous indique les principales implémentations. Certaines sont basées sur des tableaux pour stocker les éléments (`ArrayList`, `ArrayDeque`), d'autres sur des arbres (`TreeSet`, `TreeMap`), d'autres avec des tables de Hash (`HashSet`, `HashMap`). Chaque implémentation a ses propres propriétés (avec ses propres avantages et défauts). Mieux vaut donc lire la documentation pour savoir quelle implémentation est plus appropriée à chaque utilisation. Pour des utilisations « simples » (où les collections ne contiennent pas un trop grand nombre d'objets) on ne verra guère de différences entre les implémentations. Nous ne prendrons pas le temps ici de comparer toutes ces implémentations.



2.4 Ordre

La *classe* `Collections` (à ne pas confondre avec l'interface `Collection`) est une classe qui contient beaucoup de méthodes statiques pour manipuler des collections passées en paramètres. Par exemple, il existe une méthode `sort(List<T> maListe)` qui va trier une liste `maListe`. Si la liste contient des dates, la méthode va trier en ordre chronologique, si la liste contient des `String`, la liste sera trier par ordre lexicograhique. Mais si vous voulez trier des `Gaulois` par quantité de sangliers consommés par an, vous pourrez aussi facilement utiliser la méthode `sort` ! Derrière tout cela, une nouvelle interface est utilisée : l'interface `Comparable`.

Cette interface ne contient qu'une seule méthode : la méthode `public int compareTo(T o)`. Cette méthode retourne un entier négatif si l'objet est plus petit que l'objet passé en paramètre, zéro s'ils sont égaux, et un entier positif si l'objet est plus grand que l'objet passé en paramètre.

Si vous regardez la documentation des classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres, vous verrez que ces classes implémentent toutes l'interface `Comparable`. Ils ont donc une implémentation de la méthode `compareTo`.

Vous pouvez donc spécifier votre propre méthode `compareTo`. Par exemple pour la classe `Gaulois`, on pourrait écrire

```java
public class Gaulois extends Personnage implements Comparable<Gaulois>{
   String nom ;
   int quantiteSanglier ;
     ...

   public int compareTo(Gaulois ixis) {
     return this.quantiteSanglier - ixis. quantiteSanglier ;
   }
}
```

Attention, si vous essayez de trier une liste qui n'implémente pas l'interface `Comparable`, vous provoquerez une erreur (`ClassCastException`).

Chapitre 8

Classes Internes

Jusqu'ici, on a toujours considéré qu'un fichier JAVA contenait la description d'une seule classe. En fait, il est possible de définir une classe à l'intérieur d'une autre ! C'est ce qu'on appelle des classes internes.

Les méthodes d'une classe ont accès aux attributs de la classe, qu'ils soient `public` ou `private`. Il en est de même pour une classe interne, qui est considérée comme un membre de la classe qui la contient : elle a accès à tous les attributs (qu'ils soient publics ou privés) de la classe qui la contient. Cette visibilité peut être exploitée en considérant que la classe interne soit une sorte d'outil pour la classe qui la contient. Ainsi, elle a déjà accès aux attributs. Cette possibilité peut dans certains cas simplifier le développement. L'API JAVA utilise parfois des classes internes.

On distingue deux grands types de classes internes selon si la classe est `static` ou non. Quand la classe interne n'est pas `static`, on parle de *classe interne d'instance*.

Du point de vue de la classe englobante, on peut déclarer un objet de la classe interne comme n'importe quelle autre classe [1]. Il en est différemment pour des déclarations à l'extérieur de la classe englobante. Pour déclarer un objet de la classe interne à l'extérieur de la classe englobante, on ne peut pas utiliser simplement le nom seul de la classe interne, par exemple `Interne` car dans ce cas, JAVA chercherais une classe appelée `Interne` (et donc définie dans un fichier `Interne.java`). Mais comme la classe se trouve à l'intérieur d'une autre classe, il faut faire référence au nom de la classe englobante. Si cette dernière s'appelle `MaClasse`, on utilisera donc le nom `MaClasse.Interne`.

1 Classes internes d'instance

On ne peut accéder aux membres d'une classe interne que par l'intermédiaire d'une instance de la classe qui la contient (on dit aussi englobante). Dans l'exemple ci-dessous, on a une classe appelée `MaClasse` qui possède une classe interne appelée `Interne`. Pour instantier la classe interne, il faut donc avoir déjà instantié sa classe englobante, donc dans l'exemple on crée une instance `a` de la classe `MaClasse`.

Il reste donc à instancier un objet de la classe interne. La syntaxe pour la création de l'instance de la classe interne peut apparaître quelque peu surprenante : le mot clé **`new`** est accolé à l'instance de la classe englobante : on a donc `a.new Interne();`. Cela est finalement assez intuitif : on crée une classe interne à l'objet `a`.

1. dans l'espace de nom de la classe englobante, nous verrons plus tard la notion d'espace de noms

```java
public class MaClasse {
  int nbEtudiants;
  public class Interne {
    public int nbEtudiantsMax=25;
  }
}
```

```java
MaClasse a = new MaClasse();
MaClasse.Interne b=
  a.new Interne();
```

Nous allons revenir sur l'exemple de la liste chaînée. Nous avions besoin de deux classes : `ListeChainee<E>` et `Noeud<E>`. La classe `Noeud<E>` est vraiment une classe outil pour la liste chaînée. En dehors ce cette application, l'utilisation d'un noeud n'est pas très pertinente (par exemple pour représenter le noeud d'un graphe, on voudrait peut être avoir une liste de noeud pour le suivant au lieu d'un seul noeud). Donc, il serait bon que la classe `Noeud<E>` soit interne à la classe `ListeChainee<E>`. On a aussi vu des classes qui implémentait l'interface `Comparator`, par exemple si on utilise comme notion d'ordre la quantité de sangliers consommés par an, il pourrait être bon que cette classe soit interne à la classe `Gaulois`.

2 Classes internes `static`

La création d'une instance d'une classe interne `static` se fait indépendamment de la classe contenante. Ainsi, la classe interne n'a accès qu'aux membres non `static` de la classe contenante. Pour instancier une classe interne, comme celle-ci est indépendante de la classe englobante, on utilise `new MaClasse.Interne();`.

```java
public class MaClasse {
  int nbEtudiants;
  public static class Interne {
    public int nbEtudiantsMax=25;
  }
}
```

```java
MaClasse a = new Externe();
MaClasse.Interne b=
  new MaClasse.Interne();
```

3 Interfaces internes ou imbriquées

Il est aussi possible d'avoir des interfaces imbriquées (ou internes) en JAVA . Un exemple qui se trouve dans l'API est l'interface `Map<K,V>` qui modélise une relation binaire : un `Map` est une collection de couples `(clé, valeur)` ou `cle` est de type `K` et `valeur` est de type `V`. Pour modéliser les couples, JAVA utilise une interface imbriquée `Map.Entry`. On a donc l'interface suivante :

```java
public interface Map<K,E> {
  interface Entry<K,V> {
    K getKey() ;
    V getValue() ;
    V setValue(V value) ;
  }
  void clear() ;
  boolean get(K key) ;
  void put(K key, V value) ;
  Collection<V> values() ;
}
```

Contrairement aux interfaces standards, une interface imbriquée peut être déclarée `private`. Il est aussi possible d'avoir une interface imbriquée à l'intérieur d'une classe. Ces utilisations sont assez avancées et nous n'entrerons pas plus dans le détail.

Chapitre 9

Gestion des classes à l'aide de Packages (espaces de noms)

Un projet en JAVA peut comporter un nombre (parfois important) de classes. De la même façon que pour ordonner les fichiers sur un disque, on peut utiliser une structure hiérarchique de répertoires pour rendre plus facile l'accès aux classes. Une autre motivation des espaces de noms est de pouvoir garantir l'unicité des noms de classes. Par exemple, deux développeur peuvent avoir l'idée de développer une classe `Dauphine`. Tant que les classes se trouvent dans des espaces de noms différents, cela ne posera aucun problème et les deux classes pourront être utilisables.

1 Déclaration d'une classe

Pour simplifier, on va faire l'hypothèse que l'on travaille sur un projet qui se trouve dans le répertoire `ProjetJava` du disque dur. Ce répertoire constituera la racine de l'espace de noms. Si on n'utilise pas de d'espaces de noms, tous les fichiers JAVA se trouveront dans ce répertoire et le nom « qualifié » de la classe est simplement son nom.

Sinon, on trouvera une hiérarchie de répertoires et de fichiers dans le répertoire `ProjetJava`. Par exemple, supposons qu'une classe `MaPremiereClasse.java` se trouve dans le répertoire `ProjetJava/important/premier/`. Cette classe se trouvera dans l'espace de noms `important.premier` : le répertoire `ProjetJava` est la racine (que l'on indique pas), ensuite on indique le répertoire qui la contient, et on sépare le nom des répertoires par le symbole « . ». Le nom qualifié de la classe sera `important.premier.MaPremiereClasse`. Ce nom qualifié sera alors unique et on pourra toujours utiliser une classe grace à son nom qualifié.

Attention, en JAVA , les espaces de noms ne s'emboitent pas les uns dans les autres. `java.util` et `java.util.function` sont deux espaces de noms distincts avec chacun leurs classes et interfaces.

Dans le fichier source JAVA , on commence toujours à indiquer à quel espace de noms la classe appartient en commençant le fichier par la ligne `package <nom_du_package>`. Cette information est redondante si on connait l'emplacement du fichier dans le disque. Cependant, elle peut s'avérer très utile lorsqu'on lit le code dans un éditeur sans pouvoir lire le placement du fichier sur le disque. JAVA vérifiera que le fichier se trouve bien dans le répertoire corespondant (i.e. la classe en question devra se trouver dans le répertoire `ProjetJava/nom_du_package/`, sinon le compilateur indiquera une erreur).

2 Utiliser une classe d'un espace de noms

Pour utiliser une classe qui se trouve dans le même espace de noms, il suffit d'utiliser son nom simple. Prenons un exemple dans lequel l'espace de noms `important.premier` contient les classes `MaPremiereClasse` et `MaSecondeClasse`. Si on a besoin de manipuler un objet de la classe `MaPremiereClasse` dans la classe `MaSecondeClasse`, on peut simplement le référencer comme un objet de classe `MaPremiereClasse`.

Pour utiliser une classe à l'extérieur de son espace de noms, on peut toujours utiliser son nom qualifié. Par exemple, on développe une classe `MaDixiemeClasse` dans l'espace de nom `negligeable/remarque` et on a besoin d'un objet de classe `MaPremiereClasse` de l'espace `important.premier`. On peut la référencer avec son chemin absolu depuis la racine, i.e. `important.premier.MaPremiereClasse`.

Cependant, cela peut être lourd à l'écriture et JAVA propose un mécanisme pour utiliser une classe par son nom simple : c'est le rôle du mot clé `import`. Lorsqu'on a besoin plusieurs fois de référencer une classe à l'extérieur de l'espace de nom, on peut *importer* soit toutes les classes d'un espace de nom, soit une classe en particulier : l'import permet de faire comme si la/les classe(s) importée(s) font partie(s) de l'espace de nom courant.

— `import important.premier` fait comme si toutes les classes de l'espace de noms `important.premier.*` faisaient partie de l'espace de nom courant. On pourra donc maintenant utiliser les classes `MaPremiereClasse` et `MaSecondeClasse`.
— `import important.premier.MaPremiereClasse` fait comme si la classe `MaPremiereClasse` de l'espace de nom `important.premier` faisait partie de l'espace de nom courant. Dans ce cas, la classe `MaSecondeClasse` ne peut pas être utilisée comme `MaSecondeClasse` (elle peut être utilisée en faisant un autre import, ou en utilisant son chemin absolu `important.premier.MaSecondeClasse`).

L'import se fait *toujours* au début de la classe.

Attention, lorsque vous utilisez `*`, vous importez seulement les classes, pas les sous espaces de noms. Ainsi, vous ne pouvez pas utiliser `import java.*` pour obtenir tous les espaces de noms qui commencent par `java.`.

Attention aussi lorsque vous importez plusieurs espaces de noms (ce qui est fréquent), car il est possible d'avoir des conflits de noms si les espaces de noms contiennent deux classes de même nom. Si on est intéressé par une seule de ces classes, on peut importer la classe spécifique. Sinon (si on veut utiliser les deux classes), il faudra utiliser le nom qualifié complet.

L'utilisation de packages permet alors de désigner sans ambigüité une classe. On peut donc avoir deux méthodes avec exactement la même signature tant que ces deux méthodes appartiennent à des espaces de noms différents.

3 Librairies de classes

Le langage JAVA possède lui-même un ensemble de classes. Sans espace de noms, toutes ses classes se trouveraient dans un seul et même répertoire (bonjour la pagaille !). Grâce aux espaces de noms, les classes sont rangées dans une hiérarchie cohérente.

— `java.lang` contient les classes fondamentales du langage.
— `java.util` contient les classes pour manipuler des collections d'objets, des modèles d'évènements, des outils pour manipuler les dates et le temps, et beaucoup de classes utiles.
— `java.io` contient les classes relatives aux entrées et sorties
— ...

Par exemple, la classe `List` est une classe pour gérer des listes d'objets. Elle se trouve dans le package `java.util`, Evidemment, vous n'avez pas à copier à chaque fois toutes les classes définies par JAVA dans

votre répertoire de travail. L'emplacement du répertoire où se trouve l'ensemble de classes de JAVA est connu de votre système d'exploitation (windows, linux ou mac OS).

4 Compilation et Exécution

Pour compiler une classe, il faut indiquer son chemin depuis la racine.

```
javac important/premier/MaSecondeClasse.java
```

Le compilateur générera le fichier `important/premier/MaSecondeClasse.class`. Si `MaSeconde-Classe` possède une méthode `main`, on lance l'exécution en spécifiant le nom complet de la classe. Pour l'exemple, on lancerait donc

```
java important.premier.MaSecondeClasse
```

Chapitre 10

Générer de la documentation : l'outil javadoc

Lorsqu'on écrit du code, il est important de bien le commenter et le documenter. Cela permet aux autres programmeurs de comprendre votre code. Cela permet aussi de se souvenir de certaines astuces utilisée ou de justifier certaines opérations. Lorsqu'une méthode est écrite, il est ainsi bon d'expliquer succinctement ce qu'elle fait, quels sont ses arguments, etc.

L'idée est donc de placer dans le code source des commentaires formatés. Ensuite, JAVA fournit un outil qui va lire le code source et générer automatiquement de la documentation (sous la forme de pages `html`) à partir des commentaires : cet outil s'appelle `javadoc`. La documentation des classes sur le site d'oracle n'est rien d'autre que le résultat de `javadoc` sur le code source des classes JAVA .

La documentation produite par `javadoc` est destinée à l'utilisateur final des classes, et pas forcément aux développeurs de ces classes. Ainsi, par défaut, `javadoc` extrait de l'information sur les éléments `public` ou `protected`. Ainsi, au minimum, vous *devez* rédiger de la documentation pour tous ces éléments :

- les packages
- les classes et les interfaces `public`
- les variables `public` et `protected`
- les méthodes et les constructeurs `public` et `protected`

Il existe une option pour aussi générer de la documentation pour des éléments `private`, ce qui sera très utile pour les autres développeurs. En tant que développeur, mieux vaut prendre la bonne habitude de commenter tous les éléments avec des commentaires `javadoc`. On peut utiliser `javadoc` pour générer plusieurs documentations, une pour les utilisateurs, l'autre pour les développeurs.

Le commentaire `javadoc` se différencie d'un commentaire « normal » par sa balise de début de commentaire qui est `/**`. Il se termine par contre comme un commentaire normal par `*/`.

Un commentaire `javadoc` donne des informations sur l'élément qui suit dans le code source (ainsi, lorsqu'on lit le code source, on peut lire l'explication de l'élément, ce qui doit suffire à comprendre, et ensuite, on peut lire le code source si besoin).

Pour chaque commentaire `javadoc`, la première phrase sera le résumé (et sera vu comme tel par `javadoc`). Cette description sera suivie de différents champs qui varient selon le type d'élément décrit (on ne décrira pas de la même manière une classe et une méthode !). Chacun de ces champs est défini par une balise qui commencera par le caractère `@` (par exemple `@author`, `@param`, etc...).

Comme on génère un code `html`, on peut utiliser des balises `html` à l'intérieur du commentaire. Par exemple `<em>` pour l'emphase (italique), `<strong>` pour le texte en gras, `<code>` pour du code source, on peut même inclure des images, etc. Evidemment, comme l'outil `javadoc` va gérer la mise en page, n'utilisez pas des balises pour les titres (`<h1>`, `<h2>`) ou pour `<hr>` pour placer une barre horizontale !

1 Commentaires pour les classes

Après avoir expliqué le rôle de la classe dans la première phrase, deux champs peuvent être utiles :
— `@author` : qui est l'auteur du code source (on peut placer plusieurs auteurs en répétant le commentaire).
— `@version` : quelle est la version de la classe.

```
1  /** A <code>Gaulois</code> objet is a Personnage who is a Gaulois
2    * @author Goscinny
3    * @author Uderzo
4    * @version 36.1
5    */
6  public class Gaulois extends Personnage {
7     ...
8    }
```

2 Commentaires pour les méthodes

La première phrase, on le redit, résume le rôle de la méthode. Le reste du commentaire va expliquer chaque élément de la signature de la méthode : on va ainsi décrire le rôle de chaque paramètre, qu'est-ce que retourne la méthode, etc.
— Chaque paramètre de la méthode doit être expliqué. On commence par la balise `@param`, suivie du nom d'un paramètre, puis on explique le rôle du paramètre.
— On décrit ce que retourne la méthode (quand elle n'est pas `void`) après la balise`@return`
— On décrit toute exception levée après la balise `@throws`

```
1  /** tells whether the gaulois thinks whether a fish is fresh
2    * @param fish the fish in question
3    * @return true when the gaulois thinks the fish is fresh,
4    * false otherwise
4    */
5  public boolean isFresh(Fish fish){
6
7  }
```

3 Commentaires pour les variables d'instance ou de classe

Pour une utilisation par défaut, seules les variables de classes ou d'instance qui sont `public` seront extraites par `javadoc` [1]. Dans ce cas, il suffit d'utiliser un commentaire `javadoc` et aucun autre champ n'est demandé.

```
1  /** duration in time of the effect of the magic potion
2    */
3  public int magicPotionDuration ;
```

1. On rappelle que par défaut, `javadoc` génère de la documentation pour les utilisateurs des classes, pas les développeurs.

4 Autres commentaires

- `@since` décrire la version à partir de laquelle l'élément est disponible
- `@deprecated` indique que la classe, méthode ou variable ne devrait plus être utilisée
- `@see cible` va créer un lien vers la référence `cible`.
  La cible peut être :
  - une classe, une méthode ou une variable commentée
    `@see courseExamples.IrreductibleGaulois#fight(Fighter f)` va faire un lien avec la méthode `fight(Fighter f)` qui se trouve dans la classe `IrreductibleGaulois` du package `courseExamples`.
  - un lien `html`
    `@see <a href=''http ://www.asterix.com''>official webpage</a>`

5 Commentaire pour les packages

Dans tout ce qu'on a vu jusqu'ici, il suffisait d'écrire des commentaires directement dans le code source. Pour les packages, il faut écrire dans un fichier séparé dans chaque répertoire :

- un fichier `package-info.java` contient un commentaire `javadoc` décrivant le package
- on peut aussi fournir un fichier "overview" dans un fichier `overview.html`. Tout ce qui sera à l'intérieur des balises `<body> </body>` sera utilisé par `javadoc`.

6 Générer la documentation

```
javadoc -d docDirectory package1 package2
```

La commande va générer la documentation pour les packages `package1`, `package2` et la placera dans le répertoire `docDirectory`.

Si on veut faire des liens automatiquement vers la document de la librairie standard de JAVA , on peut utiliser l'option `-link`

```
javadoc -link https ://docs.oracle.com/javase/8/docs/api/ *.java
```

Chapitre 11

Archivage de classes

L'utilitaire `jar` fait parti de la bibliothèque standard et permet de faire une archive de classes.

```
jar cvf bibliotheque.jar monRepertoireJava/mesClasses/*.class
```

Cet utilitaire fonctionne comme le programme `tar` sous unix

- c create an archive
- u mise à jour de l'archive
- x extraction de l'archive
- v verbose
- f use archive file
- e point d'entrée pour une application archivée dans un fichier `jar` exécutable

```
jar cvfe pro.jar projet.classes.Principale projet/classes/*.class
```

1 Utiliser un fichier `jar`

Lorsqu'on utilise des fichiers `jar`, il faut dire au compilateur et à la machine virtuelle où se trouvent ces fichiers ! Il faut donc spécifier le `class path` qui indique où sont enregistrés les fichiers utiles pour la compilation et l'exécution du code JAVA . Le `class path` peut contenir

— des répertoire qui contiennent des fichiers `.class`
— le chemin de fichiers `jar`
— des répertoires contenant des fichiers `jar`

Le compilateur et la machine virtuelle ont une option `-classpath` (abbrégée en `-cp`)

```
java -cp . :../libs/lib1.jar projet.classes.Principale
```

Le `class path` aura deux éléments : le repertoire courant `.` et un fichier `jar` nommé `lib1.jar` dans le répertoire `../libs`.

Sous Windows, il faut utiliser des `;` pour séparer les éléments. Sinon, il faut utiliser `:`

Vous pouvez mettre à jour une variable d'environnement `CLASSPATH` sous votre sysème.

Chapitre 12

Annotations

Les annotations sont des balises dans le code JAVA que des outils peuvent utiliser. Ces annotations ne sont donc pas destinées au compilateur JAVA , on peut les voir comme des commentaires particuliers. Parmis les outils qui utilisent les annotations, on peut citer :

- JUnit : les annotations indiquent des méthodes qui exécutent un test (cf seconde partie du cours)
- JAVA Persistence Architecture : relation entre des classes et des tables d'une base de données
- Check Framework : permet par exemple d'ajouter des assertions dans le programme (ex : paramètre est non `null`) ou qu'une chaîne de caractères contient une expression régulière. Un outil d'analyse statique vérifie si les assertions sont valides dans le code.
- outils de la bibliothèque standard pour la compilation

L'objectif de ce chapitre est d'indiquer la syntaxe pour pouvoir lire un code annoté (au cas où vous en rencontriez dans un code déjà écrit). Nous n'entrerons pas dans les détails mais nous verrons une application avec Junit , qui permettra de réaliser des tests.

1 La syntaxe des annotations

Une annotation commence par le caractère `@` qui est suivi par le nom de l'annotation. En option, on peut avoir des couples (clé,valeur) appelés *éléments*. Les valeurs peuvent être

- la valeur d'un type primitif
- une chaîne de caractères
- un objet de type `Class`
- une instance d'un type énuméré
- une autre annotation
- un tableau d'un des types précédents (sauf un autre tableau !)

```
@BugReport(reportedBy={"Barack", "Michelle"})
```

On peut donc annoter des déclarations, ces annotations peuvent apparaître lors de la déclaration de

- classes et interfaces
- méthodes
- constructeurs
- variables d'instances
- variables locales
- paramètres de type
- variables de générique

— package

les annotations doivent être déclarées dans une interface d'annotations, mais il existe des annotations standards :

Annotation	Applicable à	Buts
`Override`	Méthode	Vérifie que cette méthode redéfinie une méthode d'une classe parente
`Deprecated`	toute déclaration	indique que cet élément n'est plus approuvé
`SuppressWarnings`	toute déclaration (sauf packages)	Supprimes les avertissements d'un certain type
`Generated`	toute déclaration	cet élément a été généré par un outil

Il en existe d'autres que l'ont ne verra pas ici (`@SafeVarargs`, `@FunctionalInterface`, `@Resource`, `@Resources`, `@Target`, `@Retention`, `@Documented`, `@Inherited`, `@ Repeatable`). Utiliser l'annotation `@Override` est une bonne pratique.

Une note sur `@SuppressWarnings`

L'annotation `@SuppressWarnings` supprime des messages d'avertissements pour un certain type de warnings. Si on utilise un cast qui ne peut pas être vérifié au moment de la compilation (ex : `readObject()`), le compilateur JAVA va écrire un message d'avertissement qui peut être utile.

Mais si on peut garantir qu'il n'y aura pas de problèmes, on peut ignorer cet avertissement, d'où l'utilisation de l'annotation `@SuppressWarnings`. Dans ce cas là, on peut par exemple utiliser :

```
@SuppressWarnings("unchecked")
```

Parfois la généricité ne permet pas exactement de faire ce que l'on veut (ou le code n'a pas été mis à jour depuis JAVA 5) et on peut ajouter une annotation pour confirmer que ce que l'on fait est correct.

2 Exemple d'utilisation des annotations : Tests unitaires et JUnit

L'erreur fait partie de notre nature : il faut donc *valider* tout travail. En plus, on veut éviter que les erreurs passées se reproduisent.

— tests d'intégration : le logiciel fonctionne-t-il dans son environnement d'exécution ?

— tests d'acceptation : le client accepte-t-il le logiciel ?

— tests unitaires : vérifier une "unité" du logiciel

On va se concentrer sur les tests unitaires dont le but est de tester une « unité » du code. Lorsqu'on code une unité, on peut essayer d'imaginer tous les cas de figure et prévoir le comportement correct de l'unité. Lorsqu'on veut tester l'unité, une bonne idée serait de tester ces différents cas de figue et regarder si le comportement est correct. Ceci pourrait permettre dans un premier temps d'aider à la detection des bugs. Une fois que l'unité est testée, mais que quelque chose d'autre change, utiliser ces différents cas de figure constitue aussi un bon moyen de vérifier que tout continue de fonctionner si on a fait des changements. Evidemment, il serait idéal de pouvoir automatiser les tests (et éviter d'avoir des appels à `System.out.println(...)` partout !)

Mettre en place les tests peut être couteux : réaliser des cas intéressant (quelles sont les entrées et quel est/quels sont les comportements corrects) peut prendre du temps. Si aucune erreur n'est detectée, on peut avoir l'impression d'avoir perdu du temps (mais réfléchir aux tests peut nous aider à écrire un meilleur code !). D'un autre côté, trouver un bug peut prendre beaucoup de temps.

De toute façon, il est impossible/trop coûteux de garantir qu'un code n'a aucun bug (pour cela, il faudrait faire un ensemble de tests qui couvre tous les cas possibles d'utilisation, ce qui n'est pas forcément un problème simple).

L'idée est donc de réaliser un minimum de tests vérifiant une large partie des cas d'utilisation. Cela permettra d'éviter un bon nombre de bugs sans pour autant perdre trop de temps de développement.

2.1 Première solution (à bannir) : Faire une méthode `main` pour tester la classe.

C'est l'approche naturelle. Lorsqu'on débute en programmation, on commence par faire des méthodes `main`, et on peut tester des cas d'utilisation. Cependant, le test n'est pas forcément documenté/-conservé pour pouvoir être réutilisé par la suite : on a tous écrit des méthodes `main` différents pour tester différentes méthodes, mais soit on les efface souvent au fur et à mesure, soit on les conserve, mais les tests ne sont plus forcément compréhensible au bout d'un certain temps.

2.2 Bonne pratique : écrire une classe de tests pour chaque classe développée

Notre but est d'écrire une classe qui va vérifier une large partie des cas d'utilsation. Evidemment, passer ces tests ne garantira pas l'absence d'erreurs ! Pour réaliser un cas de test, on a besoin de *trois* ingrédients

— état initial qui définit l'état des variables de l'unité de code
— état final qui définit le comportement attendu
— un outil qui vérifie le comportement attendu à celui obtenu par le code.

L'outil `Junit` va nous aider à exécuter ces tests. `Junit` ne fait pas partie de la bibliothèque standard de JAVA , c'est un outil développé pour JAVA et que l'on peut télécharger sur le site `junit.org`.

A chaque classe, on va associer une classe de test `Junit` . C'est bien une classe du language JAVA mais qui va en plus utiliser des annotations spécifiques à `Junit` . Le squelette d'une classe `Junit` est le suivant :

1. créer une instance de la classe à tester
2. générer des cas de tests :
   — générer les données en entrée
   — générer le résultat
3. appeler la méthode et *vérifier* le résultat

Lorsqu'un test échoue, on laisse une explication. Tester le bon comportement d'une unité, c'est aussi tester si les exceptions fonctionnent aussi comme attendues. On peut donc aussi tester si les exceptions sont bien jetées.

Définition de tests

— une classe de tests unitaire est associée à une classe.
— on ajoute une annotation `@Test` pour indiquer le test
— nom de la méthode quelconque
— visibilité `public`
— retour `void`
— pas de paramètre, peut lever une exception

`if ... fail`

La méthode `fail` fait échouer la méthode de test (test non validé).

static void	fail() Fails a test with no message.
static void	fail(String message) Fails a test with the given message.

On a des raccourcis d'écriture avec les méthodes `assert`.

`assert`

static void	assertTrue(boolean condition) Asserts that a condition is true.
static void	assertNull(String message, Object object) Asserts that an object is null.
static void	assertNotNull(String message, Object object) Asserts that an object is *not* null.
static void	assertSame(String message, Object expected, Object actual) Asserts that two objects refer to the same object.
static void	assertNotSame(String message, Object unexpected, Object actual) Asserts that two objects do not refer to the same object.
static void	assertEquals(double expected, double actual, double delta) Asserts that two doubles or floats are equal to within a positive delta.
static void	assertEquals(String message, Object expected, Object actual) Asserts that two objects are equal.

`@Before` et `@After`

On peut exécuter une méthode *avant chaque* test et *après* chaque test.

— `@before` par exemple pour créer les instances des états initiaux et finaux, ouvrir des fichiers pour les tests.
— `@after` pour fermer des resources ou bien effacer des fichiers tests.

`@BeforeClass` et `@AfterClass` fonctionnent de la même façon, mais la méthode est exécutée *avant* le *premier* test et *après* le *dernier* test.

Test pour vérifier le bon fonctionnement d'exceptions

On peut tester si une méthode lance bien une exception (i.e. si la méthode ne lance pas d'exception, il y a un problème).

➫ on complète l'annotation avec un élément

```
@Test(expected=ArithmeticException)
@Test(expected=expected=IllegalArgumentException.class)
@Test(expected=IllegalStateException.class)
@Test(expected=NullPointerException.class)
```

2.3 Exemple

```
package tds.td3;
import static org.junit.Assert.*;
import java.util.Random;
import org.junit.Test;

public class MyListTest1 {

   @Test
   public void sizeList() {
     MyList l = new MyList();
     l.add("toto");
     l.add("toto");
     assertEquals(2, l.size());
   }

   @Test
   public void sizeList2() {
     MyList l = new MyList();
     l.addLast("z");
     l.add("toto");
     l.add("toto");
     l.addLast("ez");

     assertEquals(4, l.size());
   }

   @Test
   public void printList() {
     MyList l = new MyList();
     l.add("toto");
     l.add("toto");
     l.add("titi");
     assertEquals("titi, toto, toto", l.toString());
   }

   @Test
   public void addList() {
     MyList l = new MyList();
     l.add("toto");
     l.add("toto");
     l.add("titi");
     l.addLast("bla");
     l.addLast("bli");
     l.add("doh");
     assertEquals("doh, titi, toto, toto, bla, bli", l.toString());
   }
```

```
53
54    @Test
55    public void addList2() {
56      MyList l = new MyList();
57      l.addLast("toto2");
58      l.add("toto");
59      l.add("titi");
60      l.addLast("bla");
61      l.addLast("bli");
62      l.add("doh");
63      assertEquals("doh, titi, toto, toto2, bla, bli", l.toString());
64    }
65
66    @Test
67    public void getList1() {
68      MyList l = new MyList();
69      l.addLast("toto2");
70      l.add("toto");
71      l.add("titi");
72      l.addLast("bla");
73      l.addLast("bli");
74      l.add("doh");
75      assertEquals("doh", l.get(0));
76    }
77
78    @Test
79    public void getList2() {
80      MyList l = new MyList();
81      l.addLast("toto2");
82      l.add("toto");
83      l.add("titi");
84      l.addLast("bla");
85      l.addLast("bli");
86      l.add("doh");
87      assertEquals("toto", l.get(2));
88    }
89
90    @Test(expected=IllegalArgumentException.class)
91    public void getInvalid() {
92      MyList l = new MyList();
93      l.addLast("toto2");
94      l.get(-1);
95    }
96
97    @Test(expected=IllegalArgumentException.class)
98    public void getInvalid2() {
99      MyList l = new MyList();
100     l.addLast("toto2");
101     l.get(10);
102   }
```

```
103
127    @Test(expected=NullPointerException.class)
128    public void sumNull() {
129      MyList l = new MyList();
130      l.add("totoo");
131      l.add(null);
132      assertEquals(5, l.sumLetter());
133    }
134
135    @Test(timeout=1000)
136    public void toStringSpeed() {
137      MyList l = new MyList();
138      for(int i=0;i<100000;i++) {
139        l.add(Integer.toString(i));
140        }
141      l.toString();
142      }
143    }
```

2.4 Exécution des tests

Les tests sont exécutés en utilisant la classe JUnitCore.

— en ligne de commande

```
java org.junit.runner.JUnitCore TestClass [... autres classes de tests ...]
```

— depuis un code JAVA

```
org.junit.runner.JUnitCore.runClasses(TestClass.class, ...)
```